

SIMULATION OF SHORTEST PATH USING A-STAR ALGORITHM

NURUL HANI NORTARJA

**FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR**

2004/2005

Abstract

The shortest path problem has been widely studied for decades. It has been applied in many practical applications. For most of the cases of the shortest route finding, the Dijkstra's algorithm is known to be an optimal search algorithm. Nonetheless, the A* algorithm do have some unique properties which is more efficient in finding the shortest path especially in human real-world problem.

In this thesis, the scope of study focuses on the efficiency of the A* algorithm to make the pathfinding for a shortest route from parking entrance to the empty parking space in parking lot in fastest time and easier way. The development of A* simulation is only one modul apart from the overall automated parking system. For WXES3181, the thesis only cover the first part of A* development and implementation that includes the algorithm flowchart, pseudocode and the calculation process of the formula for the shortest route ($F = G + H$). As for WXES3182, the thesis cover the development of programming code, the code implementation and testing and the evaluation including discussion about the problem/constraints faced in the coding program development.

This thesis also provides an explanation about the advantages, functions, characteristics, the degree of complexity in A* algorithm and its implementation in real-world application. The steps to calculate a shortest path using A* algorithm is shown by using appropriate examples and related figures. The development process for this simulation is using programming language of Microsoft Visual C++ 6.0 and the coding creation depends on the algorithm flowchart and the formula pseudocode. Besides that,

there is an interface view for the expected simulation output which mainly is in MSDos Prompt.

The A* algorithm is considered to be more efficient than Dijkstra's algorithm because it searches towards a goal without considering all nodes in a state space, but rather more focus and directed. This means A* algorithm only calculates and consider the next node in path that has the lowest value of G and F, plus searching the shortest route by using heuristic estimation in Manhattan method. Definitely this best-first search algorithm saves the searching time and improves performance of shortest pathfinding.

Acknowledgement

Utter most gratitude goes to the almighty Allah for all the confidence and patience in the completion of part 1 and 2 of the thesis (WXES3181 & WXES3182). I wish to record my indebtedness and appreciation to everyone who has been so helpful and supportive in this project work and brought it to success.

I would like to express my deep gratitude to my supervisor Mr. Yamani Mohd. Idna Idris for the tremendous help he has given me during this project, technical advice and thoughtful comment. And also to my examiner, Dr. Phang Keat Keong for his guidance and sharing his experience and knowledge. The valuable advice and motivation will be cherished thus to develop a personal values of mine in the future.

Also taking this opportunity expressing my thanks to all fellow members and especially the family of Computer Science and Networking for their constructive criticism and support to face the difficulties and challenging time.

Finally, last but not least, I am much obliged to my dear parent who have been given invaluable support and inspiration to me throughout my university life. My gratefulness also goes to all the unnamed others who directly or indirectly helped me to complete this interesting and challenging project. With this sheet of paper, I can only say thank you with all my heart.

Table of Contents

	Page
Abstract	i
Acknowledgement	iii
Table of Contents	iv
List of Tables	x
List of Figures	xi
Chapter 1 : Introduction	
1.1 Problem In Parking System	1
1.2 Automated Parking System Using A* Algorithm	1
1.3 Problem Definition	3
1.4 Objectives	3
1.5 Project Scope	4
1.6 Constraints	5
1.7 Targeted User	6
1.8 Project Schedule (Gantt Chart)	6
1.9 Summary	7
Chapter 2 : Literature Review	
2.1 Analysis of Common Parking Issues and Problems	8
2.2 Current Solution/Method	11
2.3 Proposed New Solution	11
2.4 An Automated Parking System	12
2.5 General View and Definition of A* Algorithm	13

2.6	Features of A* Algorithm	14
2.7	Why Choose A* Algorithm?	18
2.8	Existing Application Using A* Algorithm	19
2.9	Review of Pathfinding Algorithms	22
2.9.1	Graph Algorithms	22
2.9.2	Search Algorithms	27
2.9.3	Other Algorithms	28
2.10	Use of Pathfinding Algorithms	28
2.11	Summary	29

Chapter 3 : Methodology

3.1	Methodology	30
3.2	Software Process Model	30
3.3	Waterfall Model	30
3.3.1	Requirements analysis and definition	31
3.3.2	System and software design	31
3.3.3	Implementation and unit testing	31
3.3.4	Integration and system testing	31
3.3.5	Operation and maintenance	32
3.4	Methods of Collecting Information	33
3.5	Summary	35

Chapter 4 : System Analysis

4.1	System Analysis	36
4.2	System Requirements	36
4.2.1	Functional Requirements	36
4.2.2	Non-Functional Requirements	37
4.3	Run Time Requirements	38
4.3.1	Hardware Requirements	38
4.3.2	Software Requirements	39
4.3.3	Programming Language	39
4.3.4	Microsoft Foundation Class (MFC) Library	41
4.4	Summary	42

Chapter 5 : System Design

5.1	An Automated Parking System	43
5.2	Flowchart of A* Algorithm	45
5.3	A* Algorithm Pseudocode	46
5.4	A* Pathfinding Algorithm	47
5.4.1	Revision – What Is A*?	48
5.5	A* Algorithm Process	50
5.5.1	A – Simplifying The Search Area	51
5.5.2	B – Searching Shortest Path	52
5.5.3	C – Continuing Search	57
5.5.4	How It Works	58
5.6	Expected Simulation Output	63
5.7	Summary	65

Chapter 6 : System Development & Implementation

6.1	Introduction	66
6.2	Development Environment	66
6.3	Development Tools	67
6.4	Program Development	67
6.4.1	Review the program code	67
6.4.2	Design the program	70
6.4.3	Code the program	70
6.5	System Coding	70
6.5.1	Control Structure	70
6.5.2	Algorithm	71
6.5.3	Object Oriented Programming	72
6.6	Program Coding Approach	72
6.6.1	Simplicity and Clarity	72
6.6.2	Use meaningful variable names	73
6.6.3	Establish effective commenting conventions	73
6.6.4	Module	73
6.7	Simulator Module	73
6.7.1	Obtaining input	73
6.7.2	Allocate node memory	74
6.7.3	Search current node and all its successor nodes	74
6.7.4	Define A* specific parts and evaluate to make comparison	74
6.7.5	Display output	75
6.8	Program Coding	75
6.8.1	Coding Style	75

6.8.2	Debug Mechanism	76
6.8.2.1	Runtime error	77
6.8.2.2	Debugger	77
6.9	Summary	77

Chapter 7 : System Testing

7.1	Introduction	78
7.2	Testing Methodology	78
7.2.1	White-box testing	78
7.2.2	Black-box testing	78
7.3	Type of Testing	79
7.3.1	Module Testing	79
7.3.2	Integration Testing	80
7.3.3	System Testing	80
7.4	Example of Testing	81
7.5	Summary	84

Chapter 8 : System Evaluation & Conclusion

8.1	Simulator Strength	85
8.2	Systems Limitation and Constraint	85
8.3	Problem and Solution	86
8.3.1	Lack of programming experience	86
8.3.2	Development time factor	87
8.4	Future Enhancement	87
8.5	Summary	88

References

Bibliography

Appendix A : User Manual

Appendix B : Findpath Classes

Appendix C : Project Schedule – Gantt Chart

Appendix D : Study Case

University of Malaya

Table 1.1 : Project Schedule	7
Table 2.1 : List of Common Parking Issues	9
Table 2.2 : Examples Types of Graph Algorithms	23
Table 2.3 : Types of Search Algorithms	27
Table 2.4 : Examples Types of Other Algorithms	28
Table 3.1 : Advantages and Disadvantages of Waterfall Model	32
Table 5.1 : Pseudocode for A* Algorithm	46
Table 6.1 : Hardware Requirements	66
Table 6.2 : Development Software and Tools	67

Figure 2.1 (a) : GraphFormer by <u>Eric Marchesin</u>	19
Figure 2.2 (b) : GraphFormer by <u>Eric Marchesin</u>	20
Figure 2.3 : The first three steps of a pathfinding state space	21
Figure 2.4 : 8-Puzzle state space showing f,g,h scores	22
Figure 3.1 : Waterfall model	33
Figure 5.1 : How an automated parking system functional	43
Figure 5.2 : Flowchart of A* pathfinding algorithm	45
Figure 5.3 : Horizontal or Vertical square moved (Orthogonal/Non-diagonal) and Diagonal square moved	47
Figure 5.4 : Simplifying the search area	51
Figure 5.5 : Square grid of simple 2-dimensional array (column-x * row-y)	52
Figure 5.6 : All adjacent squares added to open list (green)	53
Figure 5.7 : Formula $F = G + H$	54
Figure 5.8 : Calculating G	55
Figure 5.9 : Calculating H in easier way	56
Figure 5.10 : Calculating F	57
Figure 5.11 : Check other 4 squares already in open list	59
Figure 5.12 : None of paths improved by going through current square -- don't change anything.	60
Figure 5.13 : Choose the one just bottom-right of starting square	61

Figure 5.14 : Other 3 squares, two -- on closed list (starting square & one just above current square, both highlighted in blue → IGNORE	62
Figure 5.15 : Last square, to immediate left of current square checked to see if G score lower, if go through current square to get there	63
Figure 5.16 : Expected output of A* pathfinding simulation – MS Dos prompt	64
Figure 5.17 : Example of graph output	64
Figure 5.18 : Expected output - MFC interface	65
Figure 6.1 : Program development process	67
Figure 6.3 : Parking map as resource from coding program	69
Figure 7.1 : Example output that applied using Manhattan method	81
Figure 7.2 : Parking map [20*20]	82
Figure 7.3 : Example output that applied not using Manhattan method	84

CHAPTER 1 :

CHAPTER 1 :

INTRODUCTION

INTRODUCTION

Chapter 1 : Introduction

1.1 Problem In Parking System

Malaysian Automobile Association (MAA) expected motor vehicles sales to rise by 5% to 425,000 units in 2004 after having fallen 6.9% to 405,010 units last year. Sales of passenger cars are expected to improve by 4.4% to 334,000, commercial vehicles by 8% to 55,000 and 4WDs by 4.8% to 36,000. The consequences of more vehicles not only impact congestion on the road, but also blow the accessibility of parking spaces in one building mainly shopping complexes. The limited parking spaces generate a number of difficulties not only to the management (to provide adequate parking space) also the consumer (to search for an open parking space which would lead to a frustrating atmosphere).

Because of the limited resources, most of the management did not undergo for more parking spaces since the average time the consumer spend to shop is about 2-3 hours. However, the management still seeks to find suitable approach to attract their consumer by providing them with the best service they could. Therefore, an automated parking system is proposed which will guide a vehicle owner to maneuver their means of transportation without the needs to rigorously locating the vacant space.

1.2 Automated Parking System Using A* Algorithm

The automated parking system will take similar means to jockey system where the vehicle will be maneuvered automatically to the space indicated by the database. This will provide most convenient way of parking and will avoid the hassle of finding unoccupied space which sometimes consume more time than the shopping itself. The other benefit of such approach is that the users do not need to remember where they have parked their vehicle because the system will allow the vehicle to maneuver back to the

original point. Although this approach suggests an inviting way, the system must be fully controlled.

An unfilled parking space must be detected before a vehicle is able to park. One approach is to implement sensors in each space, where the sensors will detect any object which filling the area. The second approach is by using image processing technique or to be exact image recognition technique that has been implemented in various applications. Each parking space has their own identity which is differentiate by their unique numbers. Therefore, the system will use the approach of character recognition to identify which parking space is free after the empty space detection has been done. Following the recognition process, the database will be updated to inform the vehicle where to park. The database will be prioritized according to the nearest shopping entrance where most consumers intend to park and also allows the system to predetermine the shortest route available to accelerate the process.

One of the problems in this type of navigation is to find the shortest route from the parking entrance to the empty space or goal. A* (A-Star) is a game programming algorithm that can be implemented to solve this difficulty because its fairly flexible and can be used in a wide range of contents. This algorithm utilize the path scoring equation

$$F(n) = G(n) + H(n)$$

where;

G = the movement cost from starting point to given square on the grid, following the path generated to get there.

H = the heuristic or estimated movement cost from given square to final destination.

Although A* can be designed using arbitrary graph, but this thesis will concentrate on designing using square grid so it is more easier to compute the direction and the distance from starting to the end point.

1.3 Problem Definition

Through the analysis process of the project, I discovered few problems that should be overcome which are :

- Limited references.
- Limited given time to finish the first part of the project, which is to analysis and design the project.
- The difficulties of implementing the theoretical of A* algorithm and approaches into real system because of its complicated coding.
- Inadmissible heuristics occur when overestimate the remaining distance between the current square and the target destination which resulted inaccurate shortest path.

1.4 Objectives

- To build a simulator which capable of finding shortest path and distance within weighted state space when the starting and destination point, plus the obstacles already known or given in more easier manner and faster pace.
- To develop a simulator using A-Star algorithm whereby later can be used by system developer; to be integrated and combine with other modules to complete the design of overall automated parking system.

- To help solving parking problems including the accessibility, congested and limited of parking system which becoming progressively more complicated in coming years.
- A simulator that can ease vehicle ownership difficulties and economically saving time by providing help guide in finding empty parking space. This simulator will outline each steps or path needed to be taken to reach destination in shortest way or fastest time.
- Upgraded the quality of current parking system to be more systematic and convenient. Lack of accessible parking can hurt local business and decrease the quality of life for residents.

1.5 Project Scope

The project scope determines part of the project process, which will overcome the burden of the overall simulator development. The following determined to what extend simulator of A* pathfinding algorithm would be developed :

- ❖ The simulator was developed to find shortest path in parking system from the starting point of node (parking entrance) to the target point of node (empty parking space/destination).
- ❖ There will be a constraint/obstacle along the path which are :
 - Impossible : No route can pass through this cell (might be a wall).
- ❖ Each state of node involve in this pathfinding will be represented by unique metric/value and color :
 - Start Point (parking entrance) – green
 - End Point (empty parking space/destination) – red

- Route (result) – yellow or red oval shape
- Impossible – blue

1.6 Constraints

There are some limitations/constraints may occur concerning the simulator to be developed which are as stated below :

- The simulator was developed purposely by providing a template or predefined maps. This template provide different kinds of mapping that generate its own value/metric for starting and destination point, plus the obstacles. Simulator need to be program to generate a value ($F = G + H$) in empty square grid when conducting pathfinding to find shortest path from starting point to destination point.
- A lot of time consuming to develop a program/coding that fully function by considering obstacles like Impossible state (wall) when searching the shortest path between two nodes. This program must be manageable and flexible to be modify or enhance the obstacles requirements; such as to Tough state (sandbank) or Very Tough state (vehicle blocking).
- Each square grid in simulator must at least contains pixel-size of 20x20 which develop total up to pixel-size of 200x200 for whole state space. It's important to ensure that the simulation can be display in clearly and appropriate manner.
- Until now, so far there aren't any applications that could find shortest path from parking entrance to the empty parking space which using A* pathfinding algorithm, applied in real-time parking system. Because of that, it's very hard to get a suitable references for this thesis project.

1.7 Targeted User

The A* pathfinding simulator will be implement and develop in an automated parking system. Due to that, this simulation is targeted for two types of user; management of shopping centre or commercialize building/area and consumer that frequently come for shopping and parked their vehicle in the provided shopping complex parking lot. For the time being, this simulator will be implement as a prototype for simulating purposes only.

This simulator is not appropriate for government or private sector use because usually their parking lot readily provided for every staff depending on their post ranking. This project basically to proof A* algorithm as the best pathfinding algorithm besides Dijkstra's and other pathfinding algorithms; to find shortest path from parking entrance (starting point) to empty parking space (destination/end point) in parking system, in more easier and faster manner.

1.8 Project Schedule (Gantt Chart)

The Gantt Chart represents the project planning phases that will be implemented to develop A* pathfinding simulator. This project schedule is followed to develop the simulator by fulfilling the objectives that has been proposed in this thesis. Table 1.1 shown the draft of Gantt Chart. The complete project schedule is shown in Appendix A.

Table 1.1 : Project Schedule

MONTH	7				8				9				10			
WEEK	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Early Research																
Problem Analysis																
Simulation Requirement Analysis																
Simulation Analysis																
Design																

MONTH	10				11				12				1				2				3			
WEEK	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Development and Implementation																								
Testing																								
Evaluation and Summary																								

1.9 Summary

This chapter briefly outline about the old problems faced by users that occurred in current parking system especially in Malaysia and shows how automated parking system by using A* pathfinding algorithm can tackle the problems. Then, the chapter continues by list out the problem definition and constraints which will be faced during the project implementation. Not forgetting to, all the project objectives and scope that it will be cover. The chapter ends by conclude the expected users for the project and project schedule which will be implemented to develop the simulator of A* pathfinding algorithm.

2.1. Analysis of Current Marketing Theory and Practices

With the high percentage of internet connectivity in Malaysia, plus the increasing of internet sales production through social media, provides the business a vastness and flexibility more than the use of paper. This is complemented by online transaction and access to marketing facilities easily. However, it is evident that although social media and internet have brought a lot of marketing facilities and tools, businesses with online sales are not doing particularly well.

Therefore, it is not surprising that many businesses are still struggling to make online sales, and many of them are not even using social media for marketing.

Figure 2.1 shows the current marketing theory and practices in Malaysia.

While marketing theory is changing, marketing practices are still the same. The current marketing practices are still based on the traditional marketing theory, which is based on the 4Ps (Product, Price, Promotion, and Place). The current marketing practices are still based on the traditional marketing theory, which is based on the 4Ps (Product, Price, Promotion, and Place).

Figure 2.2 shows the current marketing theory and practices in Malaysia. The current marketing theory is based on the 4Ps (Product, Price, Promotion, and Place). The current marketing practices are still based on the traditional marketing theory, which is based on the 4Ps (Product, Price, Promotion, and Place). The current marketing practices are still based on the traditional marketing theory, which is based on the 4Ps (Product, Price, Promotion, and Place).

Figure 2.3 shows the current marketing theory and practices in Malaysia. The current marketing theory is based on the 4Ps (Product, Price, Promotion, and Place). The current marketing practices are still based on the traditional marketing theory, which is based on the 4Ps (Product, Price, Promotion, and Place).

Chapter 2 : Literature Review

2.1 Analysis of Common Parking Issues and Problems

With the high percentage of vehicle ownership in Malaysia, plus the increasing of vehicle sales production through each year, parking has become a conflicting and confusing situation for lots of people. Major consequences are traffic congestion and limited accessibility parking spaces. Whether at an airport, bus stations, hectic traffic and compact local residents area or shopping centers (mainly), problems with parking are an everyday occurrence.

Therefore, if not fighting over traffic jams or risking lives over reckless drivers on the road, user get additional wrinkles from the stress of having to find a parking spot in the city or commercialize area especially shopping complex. Unless they are driving a Kancil (which incidentally is the country's smallest car), they might have a headache while wasting time in finding parking space for their car. In average, user spend nearly half an hour just to find an empty parking spaces to park their vehicle in parking lot. Lack of this accessible parking can hurt local business and decrease the quality of life for residents.

Apart from having trouble finding car park space, user also have to learn the many parking payment systems that Malaysians love to introduce. In Kuala Lumpur or Klang Valley, it is the meter system where they have to feed 50 cents, 20 cents or 10 cent coins into the parking meter to the time that they require. Failing to do so will result either the car will be towed or for most of the time, a summon will do the trick. A strange phenomenon, that some places can be Free Of Charge, like in Ipoh whilst some charges up to RM7.50 (at a minimum) for a period of 3 hours in Kuala Lumpur, such as shopping complex like Suria KLCC, Sungei Wang, BB Plaza, or Low Yat Plaza.

The following list identifies the kinds of problems that typically occur in a community regarding parking issues :

Table 2.1 : List of Common Parking Issues

- | |
|---|
| <ul style="list-style-type: none">▪ <i>Inadequate information for motorists</i> on parking availability and price. Motorists are likely to be frustrated if they expected abundant and free parking but find limited or expensive parking, or if they must spend excessive time searching for a parking space. |
| <ul style="list-style-type: none">▪ <i>Inefficient use of existing parking capacity.</i> Local zoning ordinances, building codes, and other development practices can result in an oversupply of parking spaces and an inefficient use of existing parking. |
| <ul style="list-style-type: none">▪ <i>Excessive automobile use.</i> Automobile dependency imposes many costs on society. User costs include reduced travel choices, increased vehicle and residential parking costs, and increased accident risk. External costs include increased road and parking facility costs, congestion, uncompensated accident damages, environmental degradation, negative land use impacts, and reduced mobility for non-drivers. |
| <ul style="list-style-type: none">▪ <i>Economic, environmental and aesthetic impacts of parking facilities.</i> Businesses ultimately bear the costs of unpriced parking, directly or through taxes that they must pass on to customers. Generous parking requirements can constrain business in other ways. |
| <ul style="list-style-type: none">▪ <i>Parking spaces that are an inconvenience</i> to nearby residents and businesses. |

Businesses may experience difficulty in retaining customers and residences may have a problem finding parking close to their homes.

- ***Demand for handicapped parking spaces.*** These spaces are generally located, in both garages and surface lots, as close to access ramps and curb cuts as possible.
- ***Impact of additional parking spaces*** on area traffic and local residents.
- ***Existing, severe, spillover problems.*** When all of the parking demand generated by a certain use (or group of uses) is not being accommodated on the site of those uses or within the adjacent on-street spaces.
- ***Out-of-town parking.*** The majority of vehicles parked in a residential area are from outside of the neighbourhood.
- ***Loading and unloading zones.*** Scarce parking for commercial vehicles to load or unload will cause them to block travel lanes.
- ***Inconvenient parking options.*** Parking within a reasonable walking distance (3 blocks) is hard to find during specific times of the day.
- ***Inadequate pricing methods.*** Many require motorists to prepay based on the maximum amount of time that they may be parked and the price structure used at a particular parking space. As a result, motorists often end up paying for time they don't actually use, and if they guess wrong they face a fine.
- ***Confusing parking policies.*** Regulations and fees may apply at certain times but not others. Parking subsidies may be provided to some users but not others.

- ***Difficulties with parking regulation and pricing.*** This problem can cause problems, including traffic congestion as motorists cruise for parking or stop in a traffic lane to wait for a space, and parking congestion in nearby areas.
- ***Lack of sufficient parking at event site.*** Special events can potentially disrupt traffic flow and require crowd management. Each event can generate its own unique transportation issues.
- ***Low parking turnover rate.*** This can occur when cars are parked in the same space for at least 4 hours (on average).

2.2 Current Solution/Method

At certain schedule of timing, guard or parking caretaker help monitoring the parking traffic flow and help the user to find empty parking space to park their vehicle. So far, some of the current approach implemented in parking system are parking payment system based either on meter system or hour system and the use of modern technologies such as automatic ticket dispensers and gate openers. Both methods mentioned just now didn't involved either guard or parking caretaker guidance whereby chances for user to get empty parking space depends on luck, whether it is peak or free hour, special occasion events either on holidays, working day or national festivals.

2.3 Proposed New Solution

To tackle the problem of user from wasting time in searching for an empty parking space in parking lot is by developing a simulator that can make pathfinding in a shortest route and fastest timing, starting from the parking entrance (starting point) to the

empty parking space (destination point). This simulation will be develop using A* pathfinding algorithm. The A* algorithm will calculate possible shortest distance between user and empty parking space that located nearby to the main entrance of shopping complex or commercialize building. The pathfinding simulator using A* algorithm is only one module part of overall automated parking system.

The simulator will be in a form of square grid of simple 2-dimensional array (column-x * row-y). Pixel (node) in square grid will represent the distance in real-world. Array will be used to represent node in square grid. By using the array method, size will be equally. The wide of state space for this algorithm will be based on the total size of parking lot. For example, if parking lot has total space of 200x200 meters, so the state space for simulation will be represent of pixels size 200x200.

Basically the constraints for this simulation are the possibility of inadmissible heuristic and considering of obstacle along the path, like a wall (Impossible state). Other obstacles like side-walk wall, vehicle blocking or entrance door in a real-world will also be considered as a limitation to the movement of A* simulation. The simulator will provide a template or predefined maps that readily generate value/metric for starting and destination point plus with the obstacles. The main simulation task is to search for the best shortest path between that two points.

2.4 An Automated Parking System

An automated parking system can automatically inform and show user which is the shortest path/way to find empty parking space in a fastest timing. Vehicle will be maneuvered automatically to the space indicated by the database. The benefits that can gain from this automated parking system are :

- The ability to provide the most convenient way of parking.

- Be able to avoid the hassle of finding unoccupied space which is sometimes consume more time than the shopping itself.
- Users do not need to remember where they have parked their vehicle because the automated system will allow the vehicle to maneuver back to the original point.

2.5 General View and Definition of A* Algorithm

The A* algorithm was originally proposed by Peter E. Hart, Nils J. Nilsson and Bertram Raphael in 1968 in a paper titled "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In that paper, it first introduced the concept of admissibility by using 3 possible cases; termination at a non-goal node, no termination and termination at a goal node without achieving minimum cost. The optimality of A* algorithm was also mentioned as when A* opens a node, then the optimal path to that node has already been found. In the year of 1972, it is further explained for some technical proofs.

The A* algorithm works much like the Dijkstra and best-first algorithms, only it values nodes in a different way. Each node's value is the sum of the actual cost to that node from the start and the heuristic estimate of the remaining cost from the node to the goal. In this way it combines the tracking of previous length from Dijkstra's algorithm with the heuristic estimate of the remaining path from the best-first search. The A* algorithm is guaranteed to find the shortest path as long as the heuristic estimate is admissible (an admissible heuristic is one that never overestimates). If the heuristic is inadmissible then the A* algorithms won't find the shortest path (or a path at all), but it will find a path faster and using less memory. While the heuristic must never overestimate, the closer it is to being correct the more efficient the A* algorithm will be;

in fact, the Dijkstra search is an A* search, where the heuristic is always 0. This algorithm also makes the most efficient use of the heuristic function, meaning that no other algorithm using the same heuristic will expand fewer nodes and find an optimal path, not counting tie-breaking among nodes of equal cost.

One of the problems of the A* algorithm, as well as many other pathfinding algorithms, is that they take up a large amount of memory by storing all previous nodes or all previously taken paths. There are some variations of the A* algorithm that are made to use less memory, these include the iterative deepening A* (IDA*) search and the simplified memory A* (SMA*) search. Some common heuristics for the A* algorithm are Manhattan distance (difference x plus difference y), Euclidean distance (the straight line distance), and the larger of the difference in x coordinates and the difference in y coordinates. Other variations on the A* algorithm may be accomplished by changing the ratio in which the heuristic is mixed with the distance so far, standard A* has this as a 50:50 ratio.

2.6 Features of A* Algorithm

The A* pathfinding algorithm is the mainly used and standard pathfinding algorithm in computer games. It is known and used for a long time (1968) and has proven to be very reliable and fast. The A* algorithm needs a map of locations or cells where each cell has is connected to some neighboring cells. The cost from moving from one cell to one of the neighboring cells is the movement cost for this cell. Again, this neighboring relation is easily fulfilled on square, rectangular or hexagonal grid. But this

pathfinding simulation to search for shortest route will use the square grid as the simulator state space representations.

As it is from the family of best-first search (BFS) strategy, the A^* algorithm is recognized by two special features which record its problem-solving path:

1- The first feature is the **shopping lists**: The pathfinding search on the square grid is performed by maintaining two separate lists of cell locations.

(a) The first list is called the **open list** and it contains all cells which should be investigated during the search. It is a state where the list has been generated and have had the heuristic function applied to them but which has not yet been further examined for its successors. At the beginning of the search this list holds only one item, the start location. However, during the search all locations which should be examined are added here.

(b) The second list is the **closed list**. It is a state where it has been further examined. This list holds all cells which were examined during the search. They are moved from the open to the closed list once the location is processed.

This concept lets the algorithm search a state space efficiently without considering an examined state (or node) twice or even repeatedly. It therefore minimizes the complexity¹ of the A^* algorithm, especially in time and space constraints.

¹ The complexity of an algorithm is a mathematic measurement of the efficiency of an algorithm.

During the search the algorithm examines one entry of the open list after the next (shopping list), determines the movement costs to this location and finally moves the entry from the open to the closed list. During this examination new cells, that is all neighbors of the currently examined cell are added to the open list, so that the search can continue in the next loop. This procedure is repeated as long as there are still items in the open list.

Each entry/node of the both lists should at least hold the following properties :

- The cost G for the current cell.
- The cost H for the current cell.
- The cell or actual location on the map (for example, x and y coordinates, or pointer to the cell).
- A link to the parent, for example the previously visited node. This is used to connect the path once you have find it. Follow the links from one node to the next will be your path.

This can for example be represented by the following class definition :

```
class ANode
{
    public:
        ANode *parent;
        int location;
        int cost_G;
        int cost_H;
};
```


2- The second feature is the *cost function*.

Unlike the breath-first search (BFS) and depth-first search (DFS) strategy, A* algorithm uses heuristic function to guide its direction of searching. It is more informed² than BFS and DFS; and it is guaranteed to find a least costly path from an initial node to the desired goal node with a minimum branching manner. The overall movement cost ($F(n) = G(n) + H(n)$) of the currently analysed cell is determined by two functions called G and H. The mathematical notations are described below :

- $G(n)$ = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
 - 1- Horizontal or Vertical square moved (Orthogonal/Non-diagonal) = cost 10
 - 2- Diagonal moved = cost 14.
- $H(n)$ = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which can be a bit confusing. It of course sounds hard or impossible to estimate the cost from a given location to the target location without knowing the path. The reason why it is called that is because it is a guess and it need not be correct..

² An informed search strategy uses problem specific knowledge to guide its node expansion in a state space.

- We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, sandbank, water, etc.). A valid assumption for the cost is for example to take a straight line from the current location to the target with movement cost for normal terrain (e.g. $\text{cost}=1$), so that cost is related to the geometrical distance between the two locations.

2.7 Why Choose A* Algorithm?

- Although this A* algorithm implementation will not 100% achieve the optimal solution means shortest possible path, still the pathfinding process can be done in more faster and convenient way compare to the other pathfinding algorithms.
- When combining the efficiency of heuristic together with the cost estimation from the current location, the performance of finding shortest path to destination could be improved.
- A* algorithm is easy to be implement in a real-world system such as computer games, road mapping or network routing.
- No computation is wasted since A* algorithm only evaluate and consider next nodes that has lowest G and F as possible path to the destination.
- When calculating the heuristic estimation (H) by using Euclidean distance means distance "as birds fly", H can be computed very easily using a standard vector math. As a result the calculation match with the length of path, but never over-estimate it.

2.8 Existing Application Using A* Algorithm

- *Understand graphs and A* path-finding algorithm with C# By Eric Marchesin.*

URL address : http://www.thecodeproject.com/csharp/graphs_astar.asp

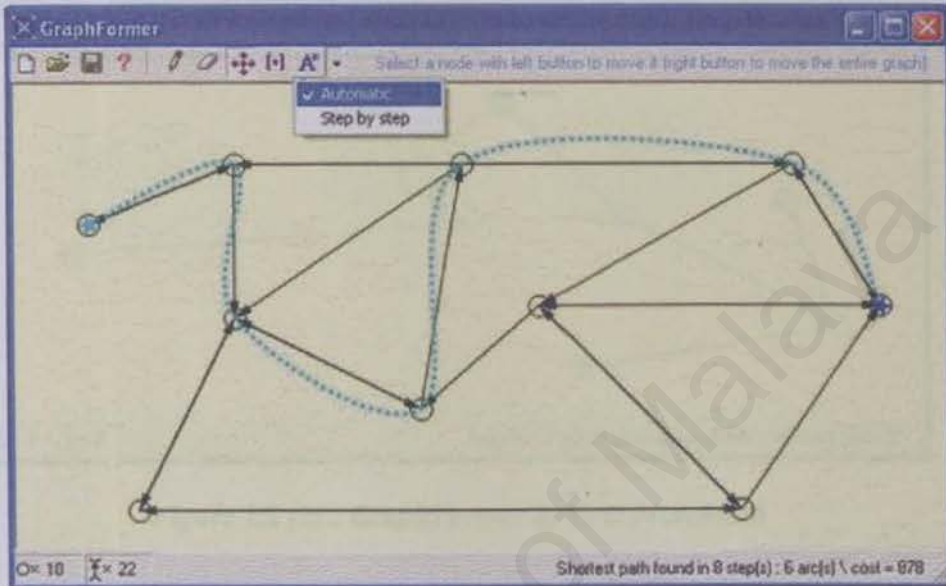


Figure 2.1 (a) : GraphFormer by Eric Marchesin

The Graph class gathers a set of methods to manage its data, such as :

- Add/Suppress a node/arc
- Get the nearest/farthest node/arc from a point
- Activate/Inactivate the entire graph
- Empty the graph

The graphical interface aims at bringing the component into play so as to reflect, fairly and simply, what it can do. The application lets you draw, move, erase or inactivate several nodes and arcs. When your graph is complete you just have to click on the 'A*' icon and select the starting and ending nodes with the respective left and right

mouse buttons. Then you will automatically see the best way. If you modify the graph, this path will be updated. If you want to visualize the algorithm's logic, then select the 'Step by Step' mode in the sub-menu of the 'A*' icon. The idea is to give the user a clear view of what happens.

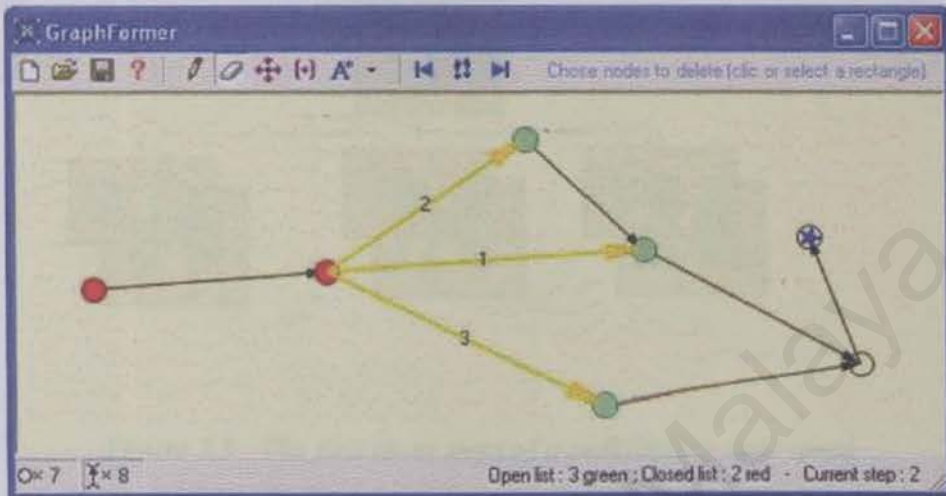


Figure 2.2 (b) : GraphFormer by Eric Marchesin

- **PathFinder2D by Intrafoundation Software.** PathFinder2D is an open-source experiment in various 2D shortest-path path-finding algorithms and techniques. This software was written in C++ using MSVC++ 6 Professional SP5 + MS Platform SDK. Algorithms involved : A*, Dijkstra, Breadth-First, Best-First and Depth-First. URL address : <http://www.intrafoundation.com/pathfinder2d.asp>

➤ *Sliding tile puzzle (the 8-puzzle) – using A* algorithm.*

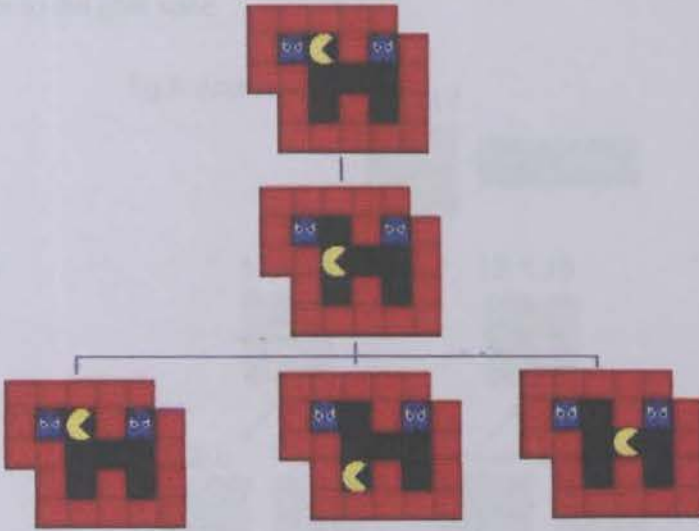


Figure 2.3 : The first three steps of a pathfinding state space

There are 362,880 different states that the puzzle can be in, and to find a solution the search has to find a route through them. From most positions of the search the number of edges (that's the blue lines) is two. That means that the number of nodes you have in each level of the search is 2^d where d is the depth. If the number of steps to solve a particular state is 18, then that's 262,144 nodes just at that level. The 8 puzzle game state is as simple as representing a list of the 9 squares and what's in them. Here are two states for example; the last one is the GOAL state, at which point we've found the solution. The first is a jumbled up example that you may start from.

Start state SPACE, A, C, H, B, D, G, F, E

Goal state A, B, C, H, SPACE, D, G, F, E

The rules that you can apply to the puzzle are also simple. If there is a blank tile above, below, to the left or to the right of a given tile, then you can move that tile into the

space. To solve the puzzle you need to find the path from the start state, through the graph down to the goal state.

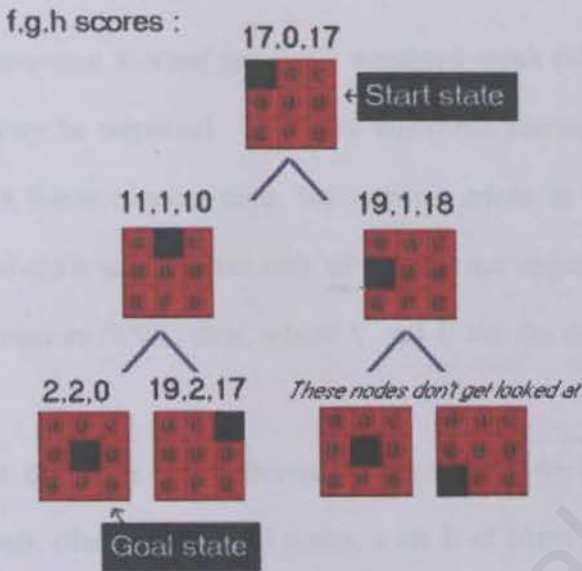


Figure 2.4 : 8-Puzzle state space showing f,g,h scores

2.9 Review of Pathfinding Algorithms

Path planning is the art of deciding which route to take, based on and expressed in terms of the current internal representation of the terrain. *Path finding* is the execution of this theoretical route, by translating the plan from the internal representation in terms of physical movement in the environment.

2.9.1 Graph Algorithms

Graph theory is the branch of mathematics that examines the properties of graphs. A graph with 6 vertices and 7 edges. Informally, a graph is a set of objects called vertices (or nodes) connected by links called edges (or arcs). Typically, a graph is depicted as a set of dots (i.e., vertices) connected by lines (i.e., edges).

Table 2.2 : Examples Types of Graph Algorithms

❖ **Bellman-Ford algorithm**

- computes single-source shortest paths in a weighted graph (where some of the edge weights may be negative). Dijkstra's algorithm accomplishes the same problem with a lower running time, but requires edges to be non-negative. Thus, Bellman-Ford is usually used only when there are negative edge weights. Bellman Ford runs in $O(VE)$ time, where V and E are the number of vertices and edges.
- In graph theory, the single source shortest path problem is the following : Given a weighted graph, (that is a set N of nodes, a set E of edges and a real-valued function $f : E \rightarrow \mathbb{R}$), and given further two elements n, n' of N , find a path P from n to n' , so that is minimal among all paths connecting n to n' . The all-pairs shortest path problem is a similar problem where we have to find such paths for every two different vertices n to n' .

❖ **Dijkstra's algorithm**

- named after its inventor, the Dutch computer scientist Edsger Dijkstra, solves the shortest path problem for a directed graph with non-negative edge weights. For example, if the vertices of the graph represent cities and edge weights represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between two cities.

❖ **Floyd-Warshall algorithm**

- In computer science, this is an algorithm to solve the All pairs shortest path

problem in a weighted, directed graph by multiplying an adjacency matrix representation of the graph multiple times. The edges may have negative weights, but no negative weight cycles. The time complexity is $\Theta(|V|^3)$. The algorithm is based on the following observation : Assuming the vertices of a directed graph G are $V = \{1, 2, 3, 4, \dots, n\}$, consider a subset $S \subseteq V$. For any pair of vertices i, j in V , consider all paths from i to j whose intermediate vertices are all drawn from S , and p is a minimum weight path from among them. The algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $S \setminus \{i, j\}$. The relationship depends on whether or not k is an intermediate vertex of path p .

❖ *Kruskal's algorithm*

- is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.
- A minimum spanning tree is a tree formed from a subset of the edges in a given undirected graph, with two properties :
 - It spans the graph - it includes every vertex in the graph.
 - It is a minimum - the total weight of all the edges is as low as possible.

In graphical form : where $w(T)$ is the minimum total weight and (u,v) is an edge between vertices u and v .

❖ *Prim's algorithm*

- is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it will only find a minimum spanning tree for one of the connected components. The algorithm was discovered in 1930 by mathematician Vojtech Jarník and later independently by computer scientist Robert Prim in 1957 and rediscovered by Dijkstra in 1959. Therefore it is sometimes called DJP algorithm or Jarník algorithm.

❖ *Borůvka's algorithm*

- is an algorithm for finding minimum spanning trees. It was first published in 1926 by Otakar Borůvka as a method of efficiently electrifying Bohemia.

Borůvka's algorithm, in pseudocode, given a graph, is :

- Copy the vertices of into a new graph, , with no edges.
- While is not connected (e.g., is a forest of more than one tree) :
- For each subtree in , find the smallest edge in connecting a vertex in to one outside it.
- Add that edge to , reducing the number of trees in by one.

❖ *Ford-Fulkerson algorithm*

- (named for L. R. Ford and D. R. Fulkerson) computes the maximum flow in a flow network.
- It works by finding a flow augmenting path in the graph. By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this computes the maximum flow in a graph. The max flow min cut theorem is a statement in optimization theory about optimal flows in networks.
- Suppose G is a finite directed graph and every edge e has a capacity $w(e)$, a non-negative real number. Further assume two vertices s and t have been distinguished. Think of G as a network of pipes; we want to pump as much stuff as possible from the source s to the sink t , never exceeding any edge's capacity.

❖ *Edmonds-Karp algorithm*

- In computer science, in the field of graph theory, this algorithm is an implementation of the Ford-Fulkerson algorithm. The important additional feature is that the shortest augmenting path is used at each step, which guarantees that the computation will terminate. In most implementations, the shortest augmenting path is found using breadth-first search.
- The Edmonds-Karp algorithm runs in $O(VE^2)$ time, where V and E is the number of vertices and edges in a graph.

2.9.2 Search Algorithms

In computer science, a search algorithm, broadly speaking, is an algorithm that takes a problem as input and returns a solution to the problem. Most of the algorithms studied by computer scientists that solve problems are kinds of search algorithms. The set of all possible solutions to a problem is called the search space. Brute-force search or "naive"/uninformed search algorithms use the simplest, most intuitive method of searching through the search space, whereas informed search algorithms use heuristics to apply knowledge about the structure of the search space to try to reduce the amount of time spent searching.

Table 2.3 : Types of Search Algorithms

Types	Function / Operation
<i>Linear search</i>	<ul style="list-style-type: none">• Finds an item in an unsorted list.
<i>Binary search</i>	<ul style="list-style-type: none">• Locates an item in a sorted list.
<i>Breadth first search</i>	<ul style="list-style-type: none">• Traverses a tree level by level.
<i>Depth first search</i>	<ul style="list-style-type: none">• Traverses a tree branch by branch.
<i>Best-first search</i>	<ul style="list-style-type: none">• Traverses a tree in the order of likely importance using a priority queue.
<i>A* tree search</i>	<ul style="list-style-type: none">• Special case of best-first search.
<i>Predictive search</i>	<ul style="list-style-type: none">• Binary like search which factors in magnitude of search term versus the high and low values in the search. Sometimes called a dictionary search.

2.9.3 Other Algorithms

Table 2.4 : Examples Types of Other Algorithms

Type	Function / Operation
<i>Hill climbing</i>	<ul style="list-style-type: none">• A graph search algorithm where the current path is extended with a successor node which is closer to the solution than the end of the current path.
<i>Genetic algorithm</i>	<ul style="list-style-type: none">• An algorithm used to find approximate solutions to difficult-to-solve problems through application of the principles of evolutionary biology to computer science.

2.10 Use of Pathfinding Algorithms

Pathfinding algorithms have many uses. These algorithms are useful in the *field of robotics*, because they can be used to guide a robot around difficult terrain without constant human intervention. This would be useful if the robot were on another planet like Mars, where some terrain must be avoided, but due to the extreme distances involved, controlling it completely via remote control would be impossible (too much delay in the radio transmission). It could also be useful if the robot were to operate underwater, where radio waves could not get to it.

Pathfinding algorithms could also be used in almost any case where a *vehicle needs to go somewhere*, while avoiding obstacles, without human intervention. Another use is in *computer games* where something needs to be moved from one place to another avoiding any walls or other obstacles in the way. These algorithms could also be used to find the *shortest way to drive between two points on a map*, the *best way to route e-mail through a computer network*, or the *shortest way to run telephone wires through existing conduits*. Some of the algorithms mentioned earlier would be better for this than others due to the fact that each one has very different characteristics and is good at different things.

2.11 Summary

This chapter basically describe the literature review more detail on problems and issues of parking, about the A* algorithm features, how the formula works and the algorithm implementation in applications; and also the background of some pathfinding algorithms.

CHAPTER 3 : METHODOLOGY

Chapter 3 : Methodology

3.1 Methodology

System development methodology is a collection of techniques for building model-applied across the simulator lifecycle. A model is process of simulator development which used for now; and later by software engineers or system developers to describe their approach in producing an automated parking system.

A software life cycle model depicts the significant phase of a simulator development from conceptions until the prototype (expected output) is completed. It specifies the relationship between project phases including transition criteria, feedback mechanism, milestones, baselines, reviews and deliverables. Typically a lifecycle model address the following phases of a simulator project; requirement phase, design phase, implementation, integration, testing, operation and maintenance.

3.2 Software Process Model

Software process is used to help software developer later to manage the simulator development, to identify what phases are present, to define the order of phases, to identify what happens in each phase and also to identify deliverable of each phase. Thus, implementation of software process in simulator development will help in producing a good simulator product.

3.3 Waterfall Model

Waterfall model was the first published model of the software development process, was derived from other engineering processes (Royce, 1970). It takes the fundamental process activities of specification, development, validation and evolution and represents them as separate process phases and because of the cascade from one

phase to another, this model is known as the “waterfall model”. It reflects engineering practice and therefore it is used widely for software development. Below are the descriptions for each phase :

3.3.1 Requirements analysis and definition

The simulator’s services (functions), constraints and goals are established by consultation with simulator users. They are then defined in detail and serve as a project specification.

3.3.2 System and software design

The simulator design process partitions the requirements to either hardware or software component. It establishes overall project architecture. Software design involves identifying and describing the fundamental simulator abstractions and their relationships.

3.3.3 Implementation and unit testing

During this stage, the simulator (software) design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

3.3.4 Integration and system testing

The individual program units or programs are integrated and tested as a complete project to ensure that the simulator requirements have been met. After testing, the simulator prototype is delivered to system developer or system engineer to be integrated with other modules to complete the overall automated parking system.

3.3.5 Operation and maintenance

Normally (although not necessarily), this is the longest life cycle phase. The simulator is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of simulator and enhancing the simulator's services as new requirements are discovered.

In principle, the result of each phase is one or more documents which approved ('signed are off'). The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other. This has caused inflexible partitioning of the project into distinct stages. Also, commitments must be made at early stage in the process and this means that it is difficult to respond to changing customer requirements. As a result, waterfall model can be used only when requirement are well-defined. Below are the concluded advantages and disadvantages of waterfall model :

Table 3.1 : Advantages and Disadvantages of Waterfall Model

Advantages	Disadvantages
✓ Discourage jumping ahead.	X Requires capturing requirements early.
✓ Emphasizes planning and good requirements.	X Makes iterative design difficult (prototyping).
✓ Testing and verification central.	X Long period before product delivery.
✓ Measurable objectives which can be used for planning future projects.	

As a conclusion, because of all advantages above and since Waterfall model only suit for smaller software products which need shorter time from requirements phase to product completion; it has influenced for this model to be chosen as a suitable methodology for simulator development using A* pathfinding algorithm. When a clear cut goal of the prototype is reached before the process begins, requirements are less likely to change.

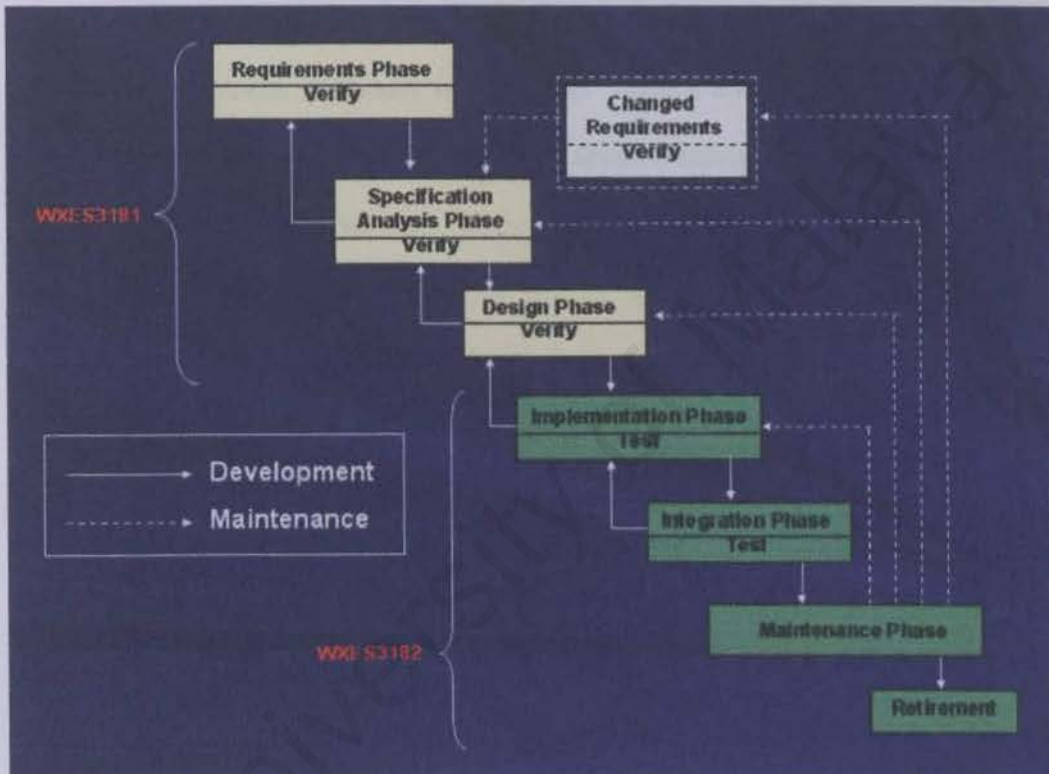


Figure 3.1 : Waterfall model

3.4 Methods of Collecting Information

There are several ways which are used to collect relevant information for this research and simulator development. The information was useful in terms of references and guidance to make the progress of simulator development running smoother. With

proper methods of collecting finding, this would be able to make the understanding clearer in terms of knowing what to do next, what is required and be updated with latest trend of pathfinding application. The methods used were reading appropriate reference books and research white paper, surfing the Internet and having discussion.

➤ **Reading**

Reading materials such as reference books, magazines, articles, dictionary and so on can be obtained from the Main Library at University of Malaya and the Document Room in Faculty of Computer Science and Information Technology (FSKTM). By referring to the previous senior's thesis, it gives ideas of how the project should be carried out.

➤ **Internet**

Information provided by the Internet is up-to-date and more concern on the recent trend of technologies. Besides, the cost to obtain knowledge from digitized information is cheaper than from information on paper.

➤ **Discussion**

Discussion was carried out together with supervisor, Mr. Mohd. Yamani Idna Idris and team member, Mohd. Yohan Ibrahim whose doing thesis on title "Shortest Route Using Dijkstra's Algorithm" plus some good hearted coursemates as to identify the aspects of system objectives, project scope, constraints, literature review on project, functional requirements, non-functional requirements, system design (in terms of algorithm, pseudo-code and flow chart) and others.

➤ **Observation and research on current parking system**

Observation has been done randomly on a few current parking systems that always full during peak hours. As a result, there aren't any implementation of electronic devices or system that can provide the needed information for the users when they navigating the vehicle from parking entrance to search for empty parking space in parking lot. It clearly means that no automated parking system yet exist and implement in Malaysia.

3.5 Summary

This chapter presents the methodology used in A* pathfinding simulator development which was Waterfall model and outline its pros and cons to be chosen. It also covers a few methods that has been used to collect information for the research purposes.

System analysis is a field of study that deals with the analysis of systems and the design of systems. It is a process of identifying the problem, defining the requirements, and designing a solution that meets the requirements. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution.

System analysis is a field of study that deals with the analysis of systems and the design of systems. It is a process of identifying the problem, defining the requirements, and designing a solution that meets the requirements. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution.

System analysis is a field of study that deals with the analysis of systems and the design of systems. It is a process of identifying the problem, defining the requirements, and designing a solution that meets the requirements. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution.

System analysis is a field of study that deals with the analysis of systems and the design of systems. It is a process of identifying the problem, defining the requirements, and designing a solution that meets the requirements. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution. The system analysis process is a continuous process that involves the identification of the problem, the definition of the requirements, the design of the solution, and the implementation of the solution.

Chapter 4 : System Analysis

4.1 System Analysis

System analysis is a part where the system requirements or specifications need to be well-identified so that the simulator can be developed up to standard and will fulfil those requirements precisely and correctly. Basically, the requirements will be categorized into two main components which are system requirements and run-time requirements.

4.2 System Requirements

System requirements are divided into two subcategories which are functional requirements and non-functional requirements. In functional requirements, all the services and functionality that can be performed by the simulator will be defined, whereas in the non-functional requirements, components that relate to this simulator properties such as its reliability, response time and flexibility will be concluded as far as possible.

4.2.1 Functional Requirements

1- Browse

This function button will load template or predefined maps which provide many different patterns of mapping that readily generated starting and destination point along with the obstacles. So far, the obstacles will be limited to Impossible state (wall) only.

2- Finding Shortest Path

This function button will make calculation of A* pathfinding algorithm, $[F = G + H]$ and searching for shortest path from starting point to destination point by considering the obstacles ahead in fastest timing.

Both function buttons above will only be implement if the MFC interface is chosen or used. Originally, the main purpose is to show the expected output of pathfinding in MSDos Prompt Windows. It will list out each next coordinate, means next point of nodes chosen in shortest path from the starting point to the destination point.

4.2.2 Non-Functional Requirements

Non-functional requirements are the other factors that must be taken into consideration in the simulator development cycle. These requirements are very subjective but they play an important role to ensure the simulator robustness and successful.

1. Reliability

The simulator should be designed in such a way that process errors in path finding are avoided or trapped before the result in output becomes error. It shall not cause any unnecessary actions of the overall environment. In simple term, the simulator must be able to convey and perform appropriate functions with minimal errors and at least 90% accurate and reliant results (the best shortest route from starting point/parking entrance to the destination point/empty parking space).

2. Manageability and flexibility

The simulator shall be capable for future expansion which to be operate, manage and integrated by the user (system developer) with other modules of sub-system to complete overall automated parking system development or with other systems and new technologies.

3. Response time

The simulator should be able to process and convey its output within a reasonable and acceptable period of time (not more than a 60 seconds).

4. Usability

The simulator must provide documentation or guideline that outline each process steps of pathfinding in detail and precise method/formula of calculation for every alternative next node in path consideration. This is for user (system developer) ease of use and manageability to conduct system's checkup if errors occur.

4.3 Run Time Requirements

Hardware and software requirements was defined as below as to run the simulator of pathfinding smoothly.

4.3.1 Hardware Requirements

- PC with a Pentium-class processor; Pentium Celeron 300 MHz or higher processor recommended.

- Microsoft Windows® 95, 98, 2000 Professional or later operating system, or Microsoft Windows NT® operating system version 4.0 with Service Pack 3 or later.
- 184 MB RAM or above.
- 1.44" Floppy disk drive or CD-ROM drive.
- VGA Monitor 14" (true color 32 bit) or higher-resolution monitor; Super VGA recommended.
- Microsoft Internet Explorer 4.01 Service Pack 1.
- Microsoft Mouse or compatible pointing device.

4.3.2 Software Requirements

- Microsoft Visual C++ 6.0 Enterprise Edition
- MSDos Prompt Windows (main choice for expected simulation output/result)
- Microsoft Foundation Class (MFC) Library (optional)

4.3.3 Programming Language

Programming language of Microsoft Visual C++ is chosen over Java for building this A* pathfinding simulator because of the factors below :

❖ Pros : Microsoft Visual C++

- ✓ Visual C++ is a very powerful, 'complete' and nice high-level language; and also as close to a universal programming language as you're likely to get at the moment. It's used everywhere.

- ✓ Visual C++ included object oriented technology which highly supported and recommended because it has a very good qualities or strengths especially in speed, performance and flexibility.
- ✓ Visual C++ boasts an Object Oriented Programming (OOP) which is very segmented, easy to work with, and doesn't require very many lines of code to perform simple tasks. By using functions and what are known as classes, certain parts of the code may be re-used multiple times throughout the program.
- ✓ Visual C++ programs are stand alone, no need for interpreters (sometimes external libraries will need to be installed on the target PC).
- ✓ Visual C++ mainly used for fast application like many desktop applications. For example, many of those run on Microsoft Windows and other Operating Systems, also in some games (such as Quake III).
- ✓ Visual C++ is one of the easiest computer languages to learn as much of the syntax is very straight-forward. However don't underestimated this programming language, as it is still extremely flexible and functional in the workforce.
- ✓ It is easy to port Visual C++ programs to other platforms if standard C++ guidelines are adhered to.
- ✓ Many Visual C++ libraries available for added functionality.
- ✓ Because of Visual C++ is a high-level language and very powerful in that, it allows the programmer benefits otherwise only available in the assembly (low-level) language. For example, programmers have much control over memory management, as can be demonstrated with arrays and linked lists.
- ✓ Visual C++ is more familiar for the author.

❖ Cons : Java Language

- X Speed. Java runs quite slow, because it is essentially compiled at run-time by the system's virtual machine. Java is a partially interpreted language, which means its code is not completely compiled into native machine code but instead compiled into byte code interpreted by the runtime environment.
- X It is difficult to compile Java programs into a stand-alone application.
- X Memory pointers not allowed in Java.
- X Java language doesn't have features like hardware-specific data types, low-level pointers to arbitrary memory addresses, or programming methods like operator overloading and multiple inheritance.
- X Some people dislike being forced into fully object oriented programming like Java language. But that's their problem.

4.3.4 Microsoft Foundation Class (MFC) Library

The Microsoft Foundation Class (MFC) Library is a collection of classes (generalized definitions used in object-oriented programming) that can be used in building application programs. The classes in the MFC Library are written in the C++ programming language. The MFC Library saves a programmer time by providing code that has already been written.

It also provides an overall framework for developing the application program. There are MFC Library classes for all graphical user interface elements (windows, frames, menus, tool bars, status bars, and so forth), for building interfaces to databases, for handling events such as messages from other applications, for handling keyboard and mouse input, and for creating ActiveX controls.

4.4 Summary

This chapter presents about the system requirements involved in A* pathfinding simulator which covers functional and non-functional requirements, also the run time requirements including hardware and software requirements that used for the simulator development. The chapter continues by briefly outline the reasons why choosing Microsoft Visual C++ as programming language to develop the simulator and describe about Microsoft Foundation Class (MFC) Library as an optional alternative for expected simulation output/result besides MSDos Prompt Windows.

CHAPTER 5
SYSTEM DESIGN

CHAPTER 5 : SYSTEM DESIGN

Chapter 5 : System Design

5.1 An Automated Parking System

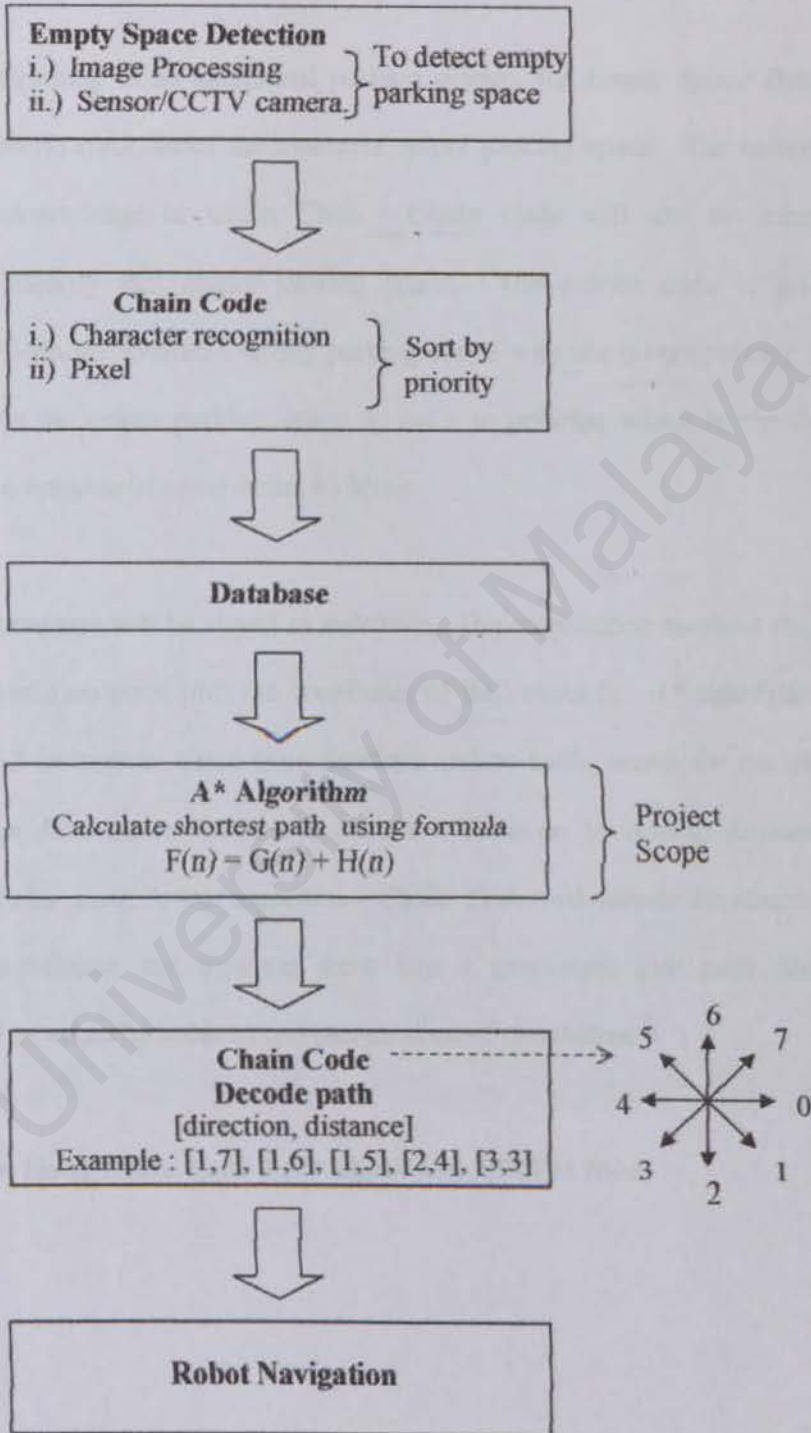


Figure 5.1 : How an automated parking system functional

Figure 5.1 shows all modules that involved besides A* pathfinding simulation to make an automated parking system perfectly function. The automated parking system will be implementing the concept of image processing, robotics and control.

At the beginning of an automated parking system, the *Empty Space Detection* will use its sensor to track down the available empty parking space. The information then will be acknowledge to Chain Code. *Chain Code* will use the *character recognition* to identify that empty parking space. The unique code is given to differentiate between the available empty parking space with the others parking space. Then, it will *sort* the empty parking space by refer to *priority*, which is the shortest distance from the entrance of commercial building.

This information will be stored in *database*. The information contains the value for start and destination point plus the coordinate of wall/obstacle. *A* simulation* will take the start and destination value from database and so forth, search for the *shortest path to the given destination*. The project scope is focus on developing simulation to find shortest path by using A-Star algorithm. *Chain Code* will *decode* the obtain result from the A* simulation and interpret them into a *movement and path distance*. Instructions will be taken by *robot* to conduct *movement simulation*.

(*Parking System Using Chain Code & A* Algorithm, FSKTM.2004*)

5.2 Flowchart of A* Algorithm

Next is the flowchart of how A* algorithm works using the formula $F(n) = G(n) + H(n)$ when searching for shortest path from starting point to the destination point..

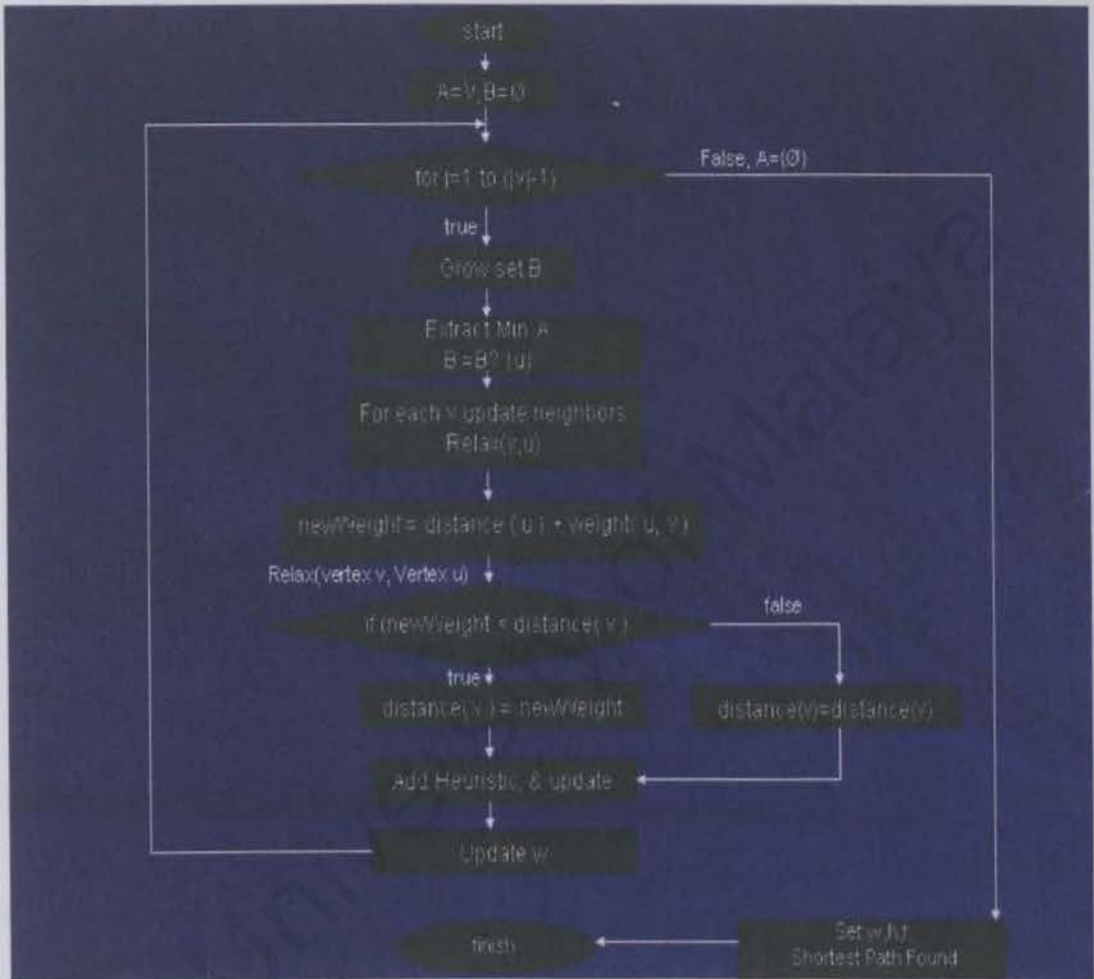


Figure 5.2 : Flowchart of A* pathfinding algorithm

5.3 A* Algorithm Pseudocode

Table 5.1 : Pseudocode For A Algorithm*

1	Create a node containing the goal state node_goal
2	Create a node containing the start state node_start
3	Put node_start on the open list
4	while the OPEN list is not empty
5	{
6	Get the node off the open list with the lowest f and call it node_current
7	if node_current is the same state as node_goal we have found the solution; break from the while loop
8	Generate each state node_successor that can come after node_current
9	for each node_successor of node_current
10	{
11	Set the cost of node_successor to be the cost of node_current plus the cost to get to node_successor from node_current
12	find node_successor on the OPEN list
13	if node_successor is on the OPEN list but the existing one is as good or better then discard this successor and continue
14	if node_successor is on the CLOSED list but the existing one is as good or better then discard this successor and continue
15	Remove occurrences of node_successor from OPEN and CLOSED
16	Set the parent of node_successor to node_current
17	Set h to be the estimated distance to node_goal (Using the heuristic function)
18	Add node_successor to the OPEN list
19	}
20	Add node_current to the CLOSED list
21	}

5.4 A* Pathfinding Algorithm

- Formula : $F(n) = G(n) + H(n)$, where
- $G(n)$ = the movement cost from the starting point to given square on the grid, following the path generated to get there.
 - 1- Horizontal or Vertical square moved (Orthogonal/Non-diagonal) equal to cost 10.
 - 2- Diagonal moved equal to cost 14.
- $H(n)$ = the heuristic or estimated movement cost from given square on the grid to the final destination. Why it is called heuristics? Because it is a guess and we really don't know the actual distance until we find the path. There's possibilities that all sorts of things can be in the way like walls, sandbank etc.

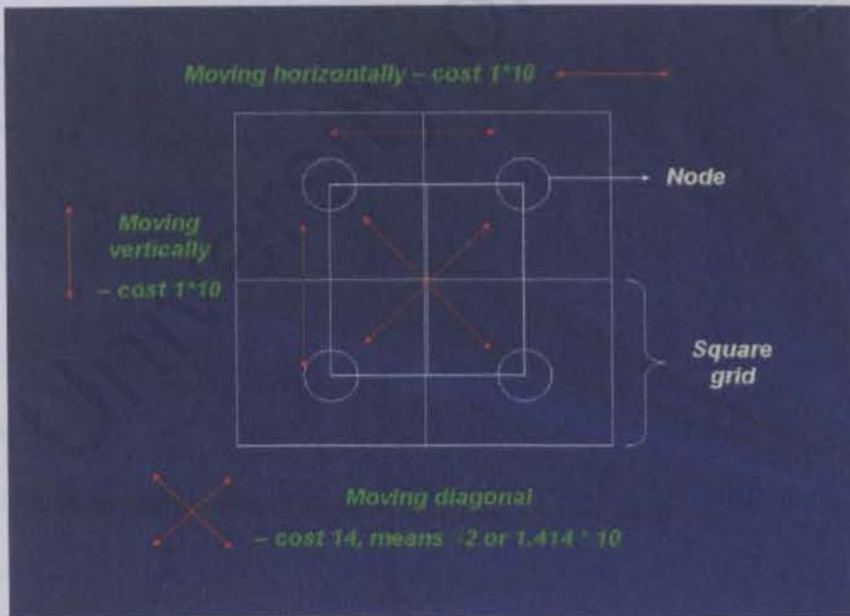


Figure 5.3 : Horizontal or Vertical square moved (Orthogonal/Non-diagonal) and Diagonal square moved

5.4.1 Revision - What Is A*?

Imagine that you are on a node and that you want to reach another position somewhere else on the graph. Then ask : "Which way will I follow, and why ?". The main factor to take into account here is the cost of moving. It must be minimal. The cost criterion is basically a function of the distance (sum of arcs' lengths). However, it can also be adjusted and varied with other data, which describe for example the slope, the harshness/practicability of the ground. You can even model a traffic jam.

To achieve the best path, there are many algorithms which are more or less effective, depending on the particular case. Efficiency depends not only on the time needed for calculation, but also on the reliability of the result. A* is able to return the best path (if it exists) between two nodes, according to accessibility/orientation and, of course, cost of arcs.

Among the variety of existing algorithms, some do not always actually return the best path, but they can give precedence to speed over accuracy. Efficiency depends on the number of nodes and on their geographical distribution. However in most cases A* turns out to be the most effective, because it combines optimized search with the use of a heuristic.

A **heuristic** is a function that associates a value with a node to gauge it. One node is considered better than another, if the final point is reached with less effort (e.g. shorter distance). A* will always return the shortest path if, and only if, the heuristic is *admissible*; that is to say, if it never overestimates. On the other hand, if the heuristic is not admissible, A* will find a path in less time and with less memory usage, but without

the absolute guaranty that it is the best one. Here are three admissible heuristics which correspond to a particular distance between the node of evaluation and the target node :

- Euclidean distance --> $\text{Sqrt}(Dx^2 + Dy^2 + Dz^2)$
- Manhattan distance --> $|Dx| + |Dy| + |Dz|$
- Maximum distance --> $\text{Max}(|Dx|, |Dy|, |Dz|)$

These functions give an estimation of the remaining distance for each node that can be explored. Thus the search can be oriented toward the 'best' nodes. For a given node, the sum [Current cost + Heuristic value] is an estimation of the cost of reaching the ending node from the starting node, passing by the current one. This value is used to continuously choose the most promising path.

In practice, the algorithm maintains 2 lists of nodes that are filled and modified during the search :

1. The first one, called **Open**, contains the tracks leading to nodes that can be explored. Initially, there is only the starting node. At each step, the best node of **Open** is taken out. Then, the best successor of this node (according to the heuristic) is added to the list as a new track. One doesn't know where the nodes that are in **Open** lead, because they have not been propagated yet. However, the best one is examined at each new step.
2. The second one, called **Closed**, stores the tracks leading to nodes that have already been explored.

5.5 A* Algorithm Process

- 1) Add the starting square to the open list.
- 2) Repeat the following :
 - a) Look for the lowest F cost square on the open list. We refer to this as the current square.
 - b) Switch it to the closed list.
 - c) For each of the 8 squares adjacent to this current square ...
 - If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.
 - If it isn't on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
 - If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.
 - d) Stop when you :
 - Add the target square to the open list, in which case the path has been found, or
 - Fail to find the target square, and the open list is empty. In this case, there is no path.

- 3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

5.5.1 A - Simplifying The Search Area

- Assume :
 - 1- Someone who wants to get from point A (parking entrance) to point B (empty parking space).
 - 2- A wall separates the two points.

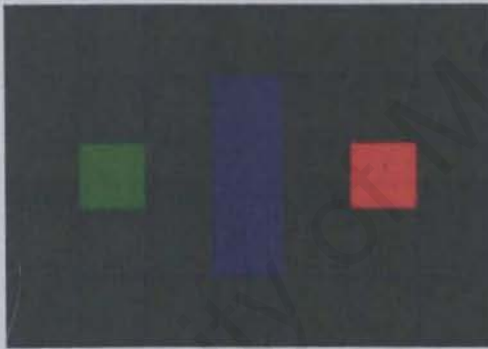


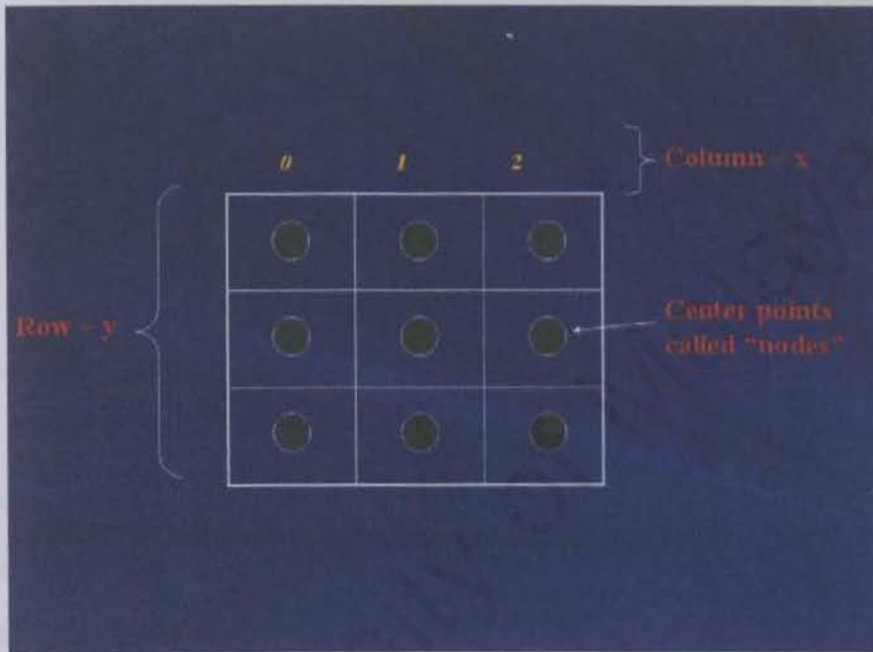
Figure 5.4 : Simplifying the search area

- Legend :

Green	= starting point A
Red	= ending point B
Blue	= wall in between 2 points.

- Simplifying the search area into square grid of simple 2-dimensional array (column-x * row-y).

- Each item in the array represents one of the squares on the grid. The grid status is recorded as either walkable or unwalkable.
- To get PATH – figure out which squares should be taken to get from A to B.
- Once path founded – person will moves from one center of one square to center of the next until the target is reached.



*Figure 5.5 : Square grid of simple 2-dimensional array (column-x * row-y)*

5.5.2 B – Searching Shortest Path

- Result :
 - 1- Green square (light blue) – starting point → indicate square has been added to closed list.
 - 2- All adjacent squares added to open list (green)
 - squares to be checked, each has gray pointer → points back to its parent (starting square).

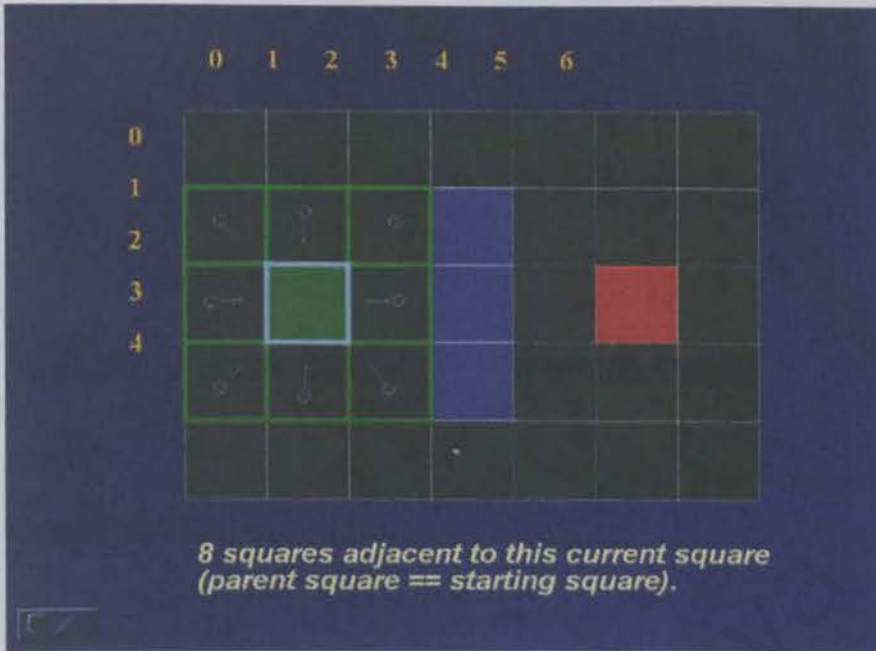


Figure 5.6 : All adjacent squares added to open list (green)

- **Path Scoring**

Path → generated by repeatedly going through open list & choosing the square with lowest F score.

- To determine which squares to use → $F(n) = G(n) + H(n)$

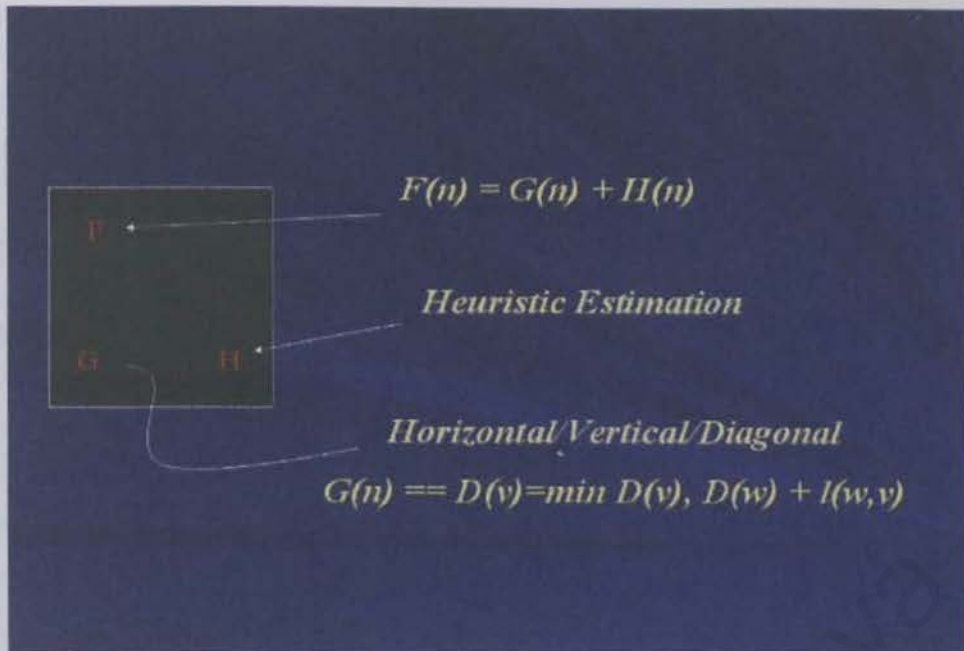


Figure 5.7 : Formula $F = G + H$

- **Calculating G**

- Cost 10 = Horizontal/Vertical square moved
- Cost 14 = Diagonal square moved
- Calculating G cost along specific path to given square → take G cost of its parent & then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square.

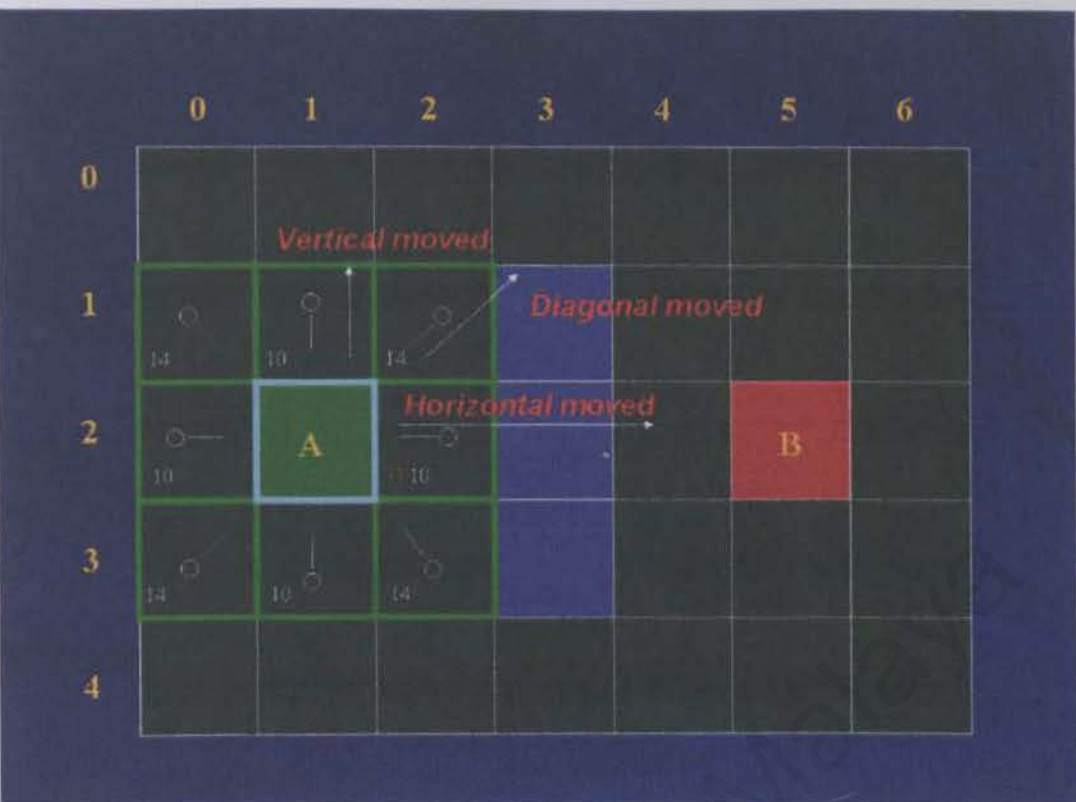


Figure 5.8 : Calculating G

• Estimating H

- Use Manhattan method -- calculate total number of squares moved horizontally and vertically to reach target square from current square, ignoring diagonal movement & ignoring any obstacles that may be in the way -- then multiply total by 10.
- Why called Manhattan? → like calculating number of city blocks from one place to another, where can't cut across block diagonally.

- **Inadmissible heuristics** – the closer estimation to actual remaining distance, the faster A* algorithm will be. If overestimate this distance, not guaranteed to give shortest path.

- **Calculating H in easier way - example :**

1- Current square (2,2) - Destination square (5,2) = (3, 0)

$3 + 0 = 3 \times 10 = 30 \rightarrow$ Reversely counting backwards from current square to destination square {30, 20, 10}

2- Current square (2,1) - Destination square (5,2) = (3, 1)

$3 + 1 = 4 \times 10 = 40 \rightarrow$ Reversely counting backwards from current square to destination square {40, 30, 20, 10}.

** Disregard the negative value.

[Previous]



Figure 5.9 : Calculating H In Easier Way

- **Calculating F**

Example : Adjacent square coordinated at (2,2);

$$F(n) = G(n) + H(n) \rightarrow 10 + 30 = 40$$

** Same rules applied to the rest of 6 squares adjacent to current square (parent square) for calculating H & F.



Figure 5.10 : Calculating F

5.5.3 C - Continuing Search

- Choose lowest F score square from all squares in open list.
- Drop selected square from open list & add it to closed list. Check all adjacent squares -- ignoring those that on closed list or unwalkable (obstacles with walls or

- sandbank), add squares to open list if they not on open list already → Make selected square the “parent” of new squares.
- If adjacent square already on open list, check whether this path to that square is better one → check to see if G score for that square is lower if we use current square to get there. If not, don’t do anything.
- If G cost of new path is lower, change parent of adjacent square to selected square.
- Finally, recalculate both F & G scores of that square.

5.5.4 How It Works

- 1- From initial 9 squares – balance 8 left on open list after starting square switched to closed list.
 - The one with lowest F cost (immediate right of starting square) -- F score 40 → select it to be next square (highlight in light blue).
- 2- Drop it from open list & add to closed list -- check adjacent squares -- the ones to immediate right of this square [wall squares] & the one to immediate left [starting square] on closed list => IGNORE
- 3- Other 4 squares already on open list -- check if paths to those squares better by using this square to get there → using G scores as point of reference. **[Figure 5.11]**
- 4- Square right above selected square -- current G score, 14.
 - If went through current square to get there – G score=20 (10, G score to get to current square + 10 more to go vertically to one just above it).
- 5- G score of 20 > 14 → not a better path -- more direct moving one square diagonally to get to that square from starting square -- rather than moving horizontally one square, and then vertically one square.

- 6- Repeat this process for all 4 adjacent squares (in open list) → none of paths improved by going through current square -- don't change anything.
- Done with this square -- move to next square. [Figure 5.11] [Figure 5.12]

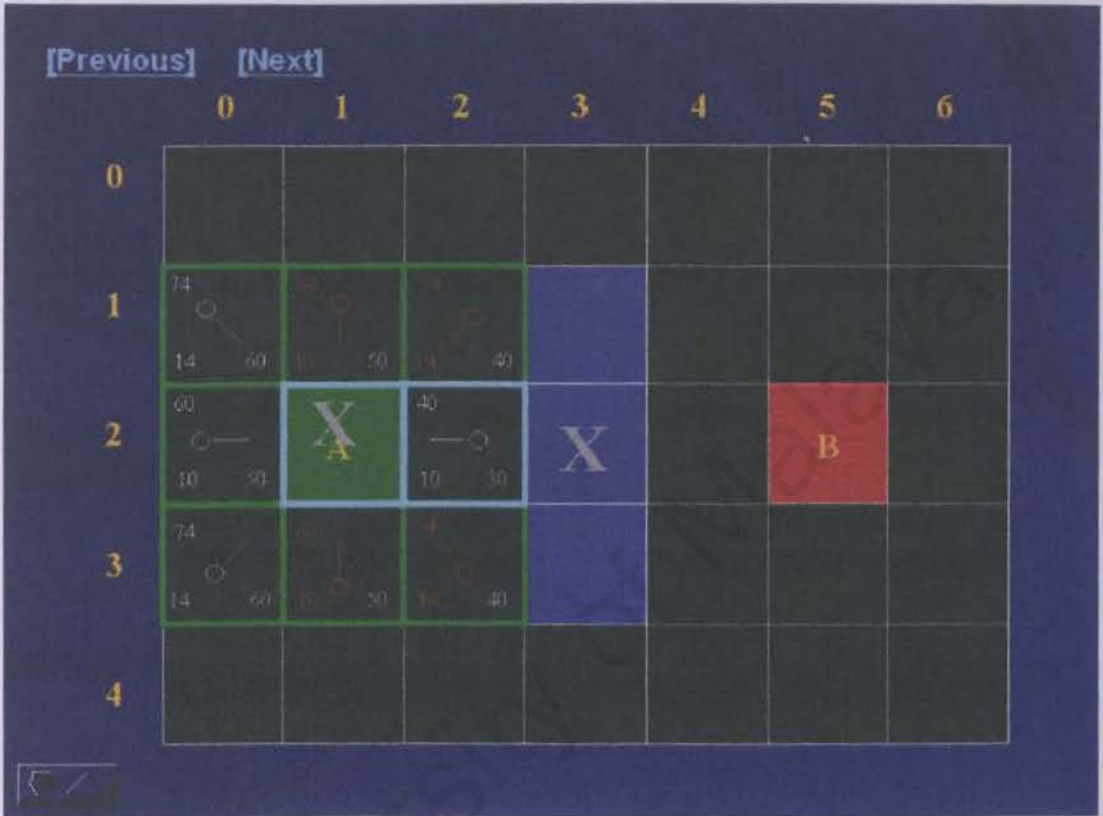


Figure 5.11 : Check other 4 squares already in open list



Figure 5.12 : None of paths improved by going through current square – don't change anything.

- 7- Go through list of squares on open list -- now down to 7 squares.
- 8- Pick the one with lowest F cost -- two squares with score 54.
 - Which do we choose? -- doesn't really matter. (Differing treatment of ties is why two versions of A* may find different paths of equal length).
- 9- Result : choose the one just bottom-right of starting square. [Figure 5.13]

[Previous]

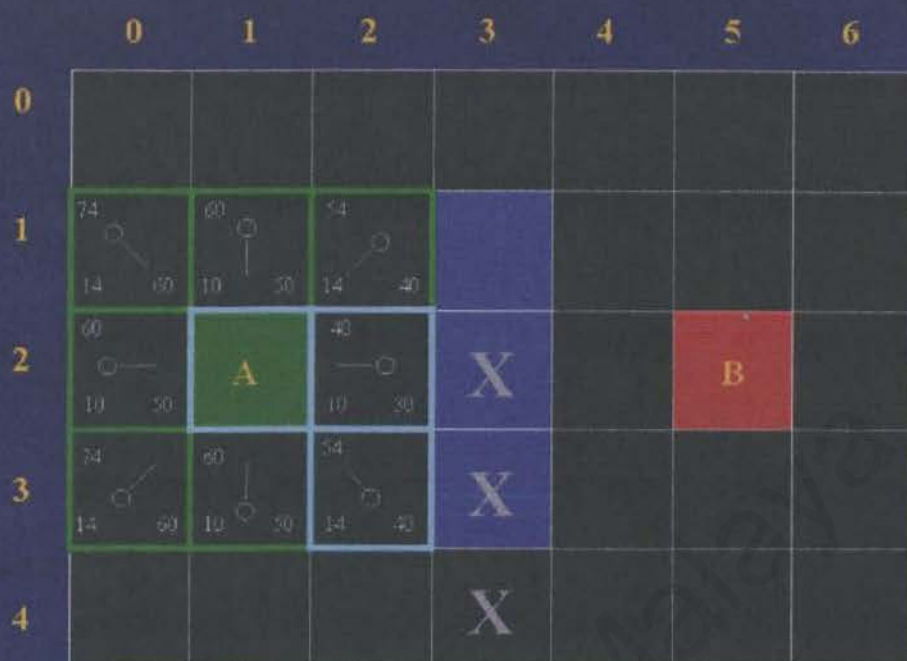


Figure 5.13 : Choose the one just bottom-right of starting square

10- Check adjacent squares. The one to immediate right & the one just above that [wall square] + the square just below the wall => IGNORE

- Why? → can't get to that square directly from current square without cutting across corner of nearby wall -- need to go down first, then move over to that square, moving around the corner in the process.

10- Leaves 5 other squares.

- Other two squares below current square aren't already on open list -- add them and current square becomes their parent.

- Other 3 squares, two -- on closed list (starting square & one just above current square, both highlighted in blue → IGNORE [Figure 5.14]

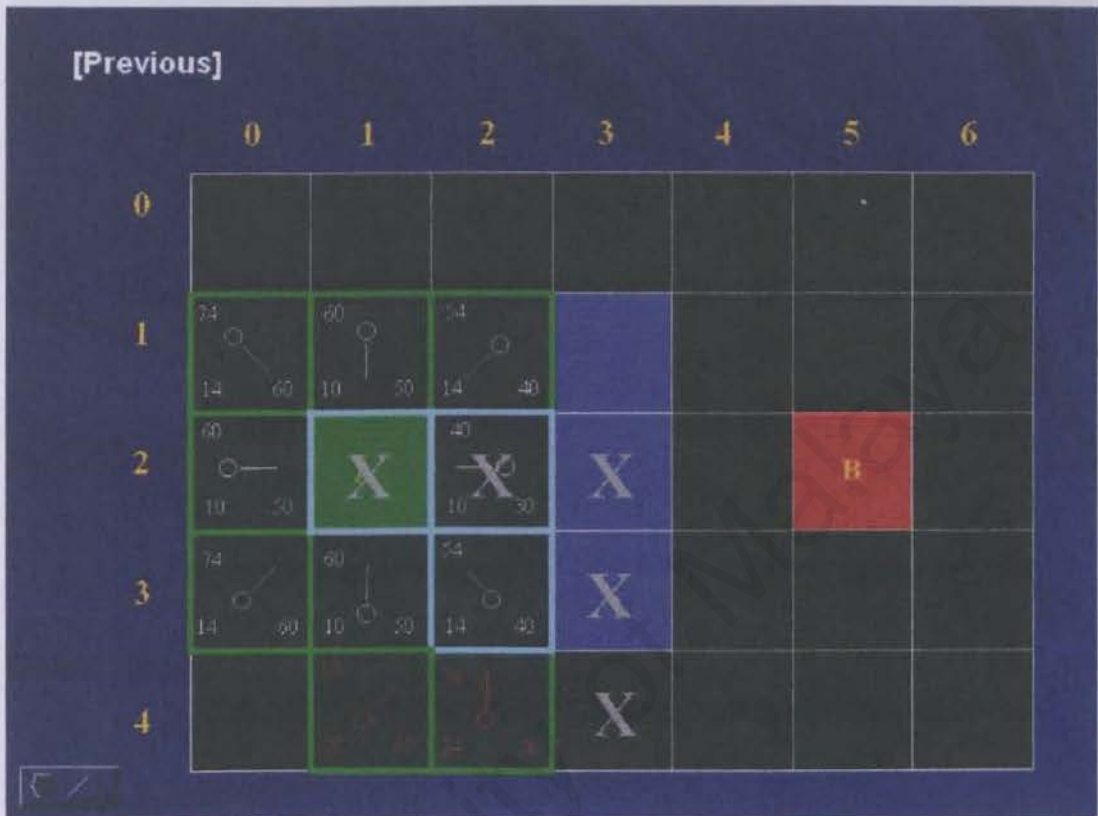


Figure 5.14 : Other 3 squares, two -- on closed list (starting square & one just above current square, both highlighted in blue → IGNORE

- Last square, to immediate left of current square -- checked to see if G score lower, if go through current square to get there → no dice -- done & ready to check next square on open list. [Figure 5.15]

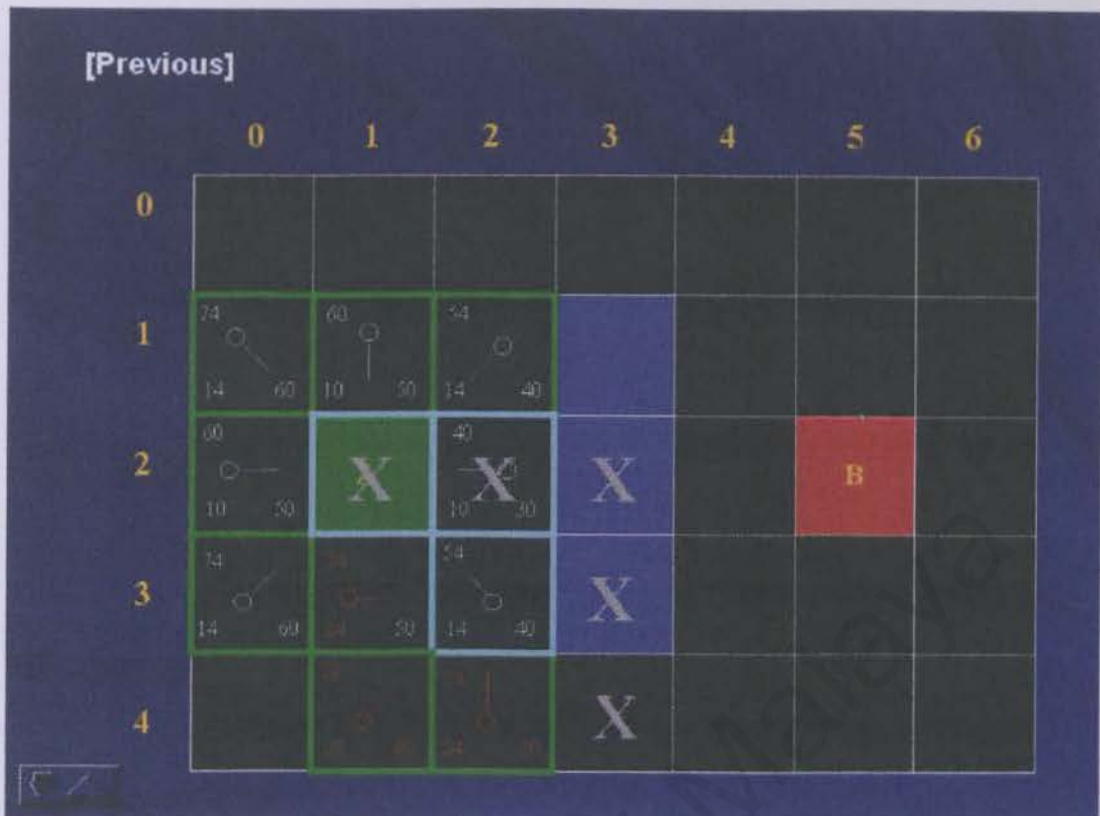


Figure 5.15 : Last square, to immediate left of current square – checked to see if G score lower, if go through current square to get there.

REPEAT SAME PROCESS UNTIL ADD TARGET SQUARE TO OPEN LIST !!!

5.6 Expected Simulation Output

The goal is to show the output in MsDos Prompt Windows rather in the MFC interface. The reason is because this simulation is target to be integrated in the hardware implementation.

```

D:\astar\Debug\astar.exe
A* path finding simulation
Output of shortest path:
[1,2][2,2],[3,2],[4,2],[5,2]
Press any key to continue

```

Figure 5.16 : Expected output of A* pathfinding simulation –MSDos prompt

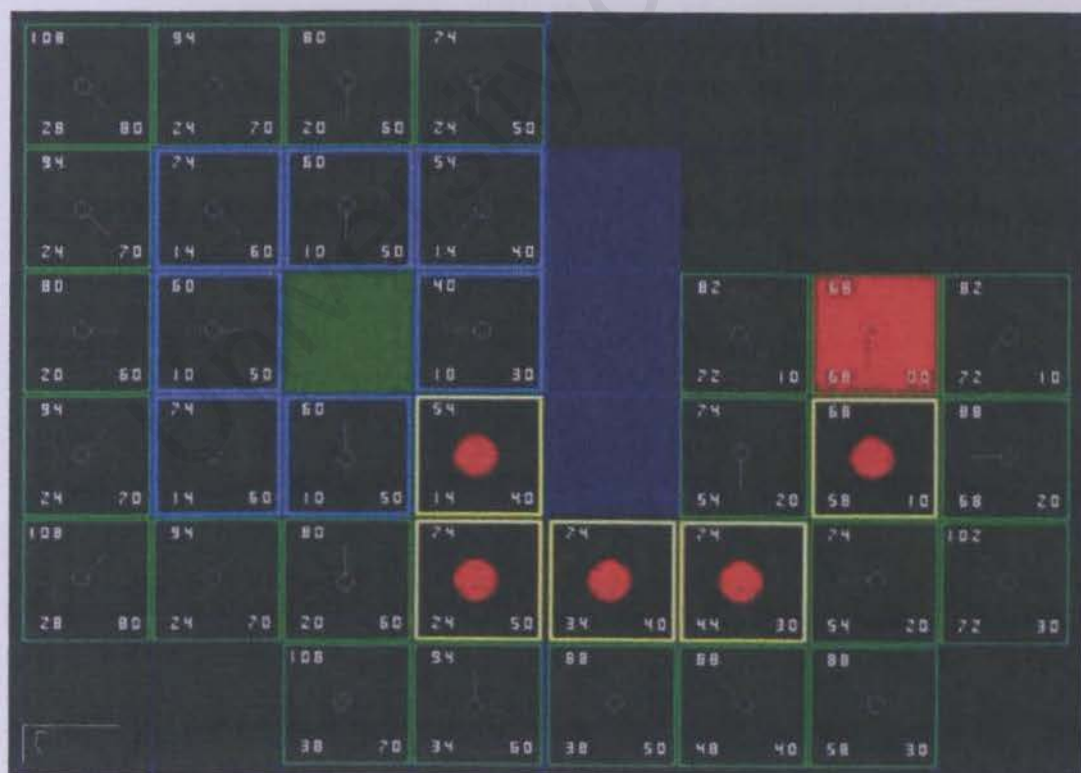


Figure 5.17 : Example of graph output



Figure 5.18 : Expected Output - MFC Interface

5.7 Summary

This chapter cover the context about all modules that are involved in the automated parking system besides the simulation of using A* algorithm. The chapter continues by describing the flowchart and pseudocode for A* pathfinding algorithm. It also outlines each step and level of how the algorithm works, plus with the expected simulation output which mainly are in MSDos Prompt Windows. The MFC interface is only the alternative which is optional for the expected simulation output.

CHAPTER 6 :

SYSTEM DEVELOPMENT

& IMPLEMENTATION

document consists of data flow of the simulator and the connection of module. The program document is then analyzed through these following steps :

- In written form, a complete definition of the requirements of the program.
- Understanding the written definition well enough to produce the desired result manually.
- Defining the input required to produce the desired output.
- Identifying the source of the input.

Generally, the first area to analyze should be the output area of the program. This comes from the written definition of the requirements. The simulator's output will be show in the MS Dos Prompt Windows (main choice for expected simulation output/result) as a layout of on the screen, showing the information that should result from program running correctly.

The second area to look at is input → parking map that consist weighted coordinate (column-x * row-y) for each point of nodes in square grid. Determine what facts are needed to produce the require information, and where that data is going to come from.

```

// Global data
// The parking space map
const int MAP_WIDTH = 20; // Represent x->column
const int MAP_HEIGHT = 20; // Represent y->row

int map[ MAP_WIDTH * MAP_HEIGHT ] =
{
// x->column
// 0001020304050607080910111213141516171819
    1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, // 00 // y->row
    1,9,9,1,9,9,1,1,1,9,1,9,9,9,9,1,1,1,1,1, // 01
    1,1,9,1,1,9,9,9,1,9,1,9,1,9,9,9,1,1,1,1, // 02
    1,9,9,1,1,9,9,9,1,9,1,9,1,9,9,9,1,1,1,1, // 03
    1,1,1,1,1,1,9,9,1,9,1,9,1,1,1,1,9,9,1,1, // 04
    1,1,1,1,9,1,1,1,1,9,1,1,1,1,9,1,1,1,1,1, // 05
    1,9,9,9,9,1,1,1,1,1,1,9,9,9,9,1,1,1,1,1, // 06
    1,1,9,9,9,9,9,9,1,1,1,9,9,9,9,9,9,1,1,1, // 07
    1,9,1,1,1,1,1,1,1,1,9,1,1,1,1,1,1,1,1,1, // 08
    1,1,1,9,9,9,9,9,9,1,1,9,9,9,9,9,9,1,1,1, // 09
    1,1,1,1,1,1,9,1,1,9,1,1,1,1,1,1,1,1,1,1, // 10
    1,9,9,9,9,9,1,9,1,9,1,9,9,9,9,9,1,1,1,1, // 11
    1,9,1,9,1,9,9,9,1,9,1,9,1,9,1,9,9,9,1,1, // 12
    1,9,1,9,1,9,9,9,1,9,1,9,1,9,1,9,9,9,1,1, // 13
    1,9,1,1,1,1,9,9,1,9,1,9,1,1,1,1,9,9,1,1, // 14
    1,1,1,1,9,1,1,1,1,9,1,1,1,1,9,1,1,1,1,1, // 15
    1,9,9,9,9,1,1,1,1,1,1,9,9,9,9,1,1,1,1,1, // 16
    1,1,9,9,9,9,9,9,1,1,1,9,9,9,9,9,9,9,1,1, // 17
    1,9,1,1,1,1,1,1,1,1,1,9,1,1,1,1,1,1,1,1, // 18
    1,1,9,9,9,9,9,9,9,1,1,9,9,9,9,9,9,1,1,1, // 19
};

```

Figure 6.3 : Parking map as resource for coding program

The last step to analyze is process. To determine what to be done to each input to turn it into output information.

6.4.2 Design the program

For the second level of program development, decisions have to be made on how the program can accomplish its tasks by developing a logical capturing solution to those program documents. The easiest way is to break the project into small pieces so and design the logic for each part of the problem.

6.4.3 Code the program

Coding programs is the process of translating the program design into the appropriate Microsoft Visual C++ 6.0 language to solve the problem. The activities in this process produce program modules that compile, build and run smoothly. Implementation of testing and analysis on the modules is to test its effectiveness and free of any error that could lead to simulator failure and malfunction.

6.5 System Coding

In system coding, every component of the program will look into this three aspects :

6.5.1 Control Structure

The control structure for the component proposed in the system design phase is translated into code. The program design structure must reflex with the control structure design. In this project the coding is done using the bottom-up approach.

6.5.2 Algorithm

The simulator program code were designed based on a specific algorithm. Algorithm is a detail sequence of actions to perform to accomplish some task. An algorithm must reach a result after a finite number of steps.

The program were broken into several steps :

- Create/define start node and goal/end node through user input.
- Allocate node memory management for start node and end node. Same process will be implement for current/parent node and all their successor nodes.
- Make advances search for each current node and find its successors nodes and all their possible moves until reach to the end node.
- Initialise value of A* specific parts : g, h, f and parent for start node and end node. Same process will be implement for current/parent node and all their successor nodes.
- Evaluate and compare each current/parent and their successors nodes for A* specific parts values : g, h, f and parent value by using push/pop method into open/close list.
- Choose nodes with lowest g value and compare their f value through class HeapCompare_f.
- Show node position (x,y) in path taken, number of solution steps and search steps taken in output, MS Dos prompt windows.

6.5.3 Object Oriented Programming

Object Oriented Programming supports object technology. It is an evolutionary form of modular programming with more formal rules that allow pieces of software to be reused and interchanged between programs. OOP is thought to increase productivity by allowing programmers to focus on higher-level software objects. One of primary features of object-oriented is inheritance. In this project, the simulator's program code was written in one source file (.cpp), two header files (.h) and one workspace file (findpath.dsw) : findpath.cpp consist three globals – GetMap(int x, int y), main(int argc, char *argv[]), map and also classes, MapSearchNode. stlstar.h contain function and classes for manipulating data in Standard Template Library (STL) - <algorithm>, <set>, <vector> and also classes AStarSearch<class UserState>. fsa.h contain FixedSizeAllocator class that used as fast fixed size memory allocator for fast node memory management.

6.6 Program Coding Approach

Factors to be taken into account when doing system coding :

6.6.1 Simplicity and Clarity

More than a few misguided programmers believe that the more complex and convoluted their code, the more sophisticated their skills. A good program is generally quite simple. The underlying meaning of the procedure represented in programming language source code should be easy to understand and clear for the programmer.

6.6.2 Use meaningful variable names

In general, variables and data structures should be named in a manner that enables the programmer to infer their meaning within the context of the procedure at hand and their correlation with some real-world object.

6.6.3 Establish effective commenting conventions

- Start with an effective prologue.
- Describe blocks of code, rather than commenting every line.
- Use blank lines and indenting so that comments can be readily distinguished from code.

6.6.4 Module

Separate function structure so it can function independently and easy for modifications.

6.7 Simulator Module

The simulator's module is divided into :

6.7.1 Obtaining input

The simulator obtain input from :

- `int map[MAP_WIDTH*MAP_HEIGHT]={};`
 - Parking map as resource in `findpath.cpp` which can be access by using map helper functions, `GetMap(int x, int y)`.
- Input by user in `main(int argc, char *argv[])` function for start and end node metric (x,y).

6.7.2 Allocate node memory

- SetStartAndGoalStates(UserState &Start, UserState &Goal)
 - By using AStarSearch<class UserState> in stlstar.h header file, assigned start and end node for memory node allocation through AllocateNode() and FixedSizeAllocator<class USER_TYPE> in fsa.h header file. Same process will be implement for current/parent node and all their successor nodes along the process of push (go in) and pop (take out) from open and close list.

6.7.3 Search current node and all its successor nodes

- The simulator will start searching the current node and all its successor nodes that possibly be the next best path to the end node by using function SearchStep(). AddSuccessor(UserState &State) is a list of successors which will be called when user expanding the search frontier and need to add successor.

6.7.4 Define A* specific parts and evaluate to make comparison

- Through public class Node in AStarSearch<class UserState> values of A* specific parts : g, h, f and parent for start node and end node will be initialised. Same process will be implement for current/parent node and all their successor nodes.
- GetSuccessors(AstarSearch<MapSearchNode>*astarsearch, MapSearchNode *parent_node) and class HeapCompare_f will be called to compare the lowest g and f value among current and its

successor nodes, and so determine next best path to be taken until reach the end node.

- GoalDistanceEstimate(MapSearchNode &nodeGoal) function is to calculate heuristics that estimates the distance from a node to the goal.

6.7.5 Display output

- By activate the debugging mechanism : #define DEBUG_LISTS 1 and #define DEBUG_LIST_LENGTHS_ONLY 1, each steps taken when push/pop from open/close list is shown by calling the functions :
GetOpenListStart(float &f, float &g, float &h),
GetOpenListNext(float &f, float &g, float &h),
GetClosedListStart(float &f, float &g, float &h) and
GetClosedListNext(float &f, float &g, float &h).
- When search found the goal state, PrintNodeInfo() will show output for each node position taken in the best path together with the solution steps and search steps by calling function GetSolutionStart() and GetSolutionNext().
- FreeSolutionNodes() function is called to clean up all used node memory when done searching.

6.8 Program Coding

6.8.1 Coding Style

There are two standard methods of program design : the top-down approach and the bottom-up approach.

- *Top-down programming* involves writing code that calls functions that haven't defined and working through the general algorithm before writing the functions that do the processing. Top-down programming is, to a good degree, a very abstract way of writing code because it starts out by using functions that haven't been designed.
- The *bottom-up approach* to programming is the opposite: writes the basic functions, then work up to the more complex parts of the program.

It's interesting that both of these approaches focus on the actions of the program rather than the objects the program manipulates - variables. Many times, the best way to write a program is to figure out the variables that need to work. By defining variables first and then working with functions that work on them, this always maintain a basic foundation of what the program should be doing. Finally, the code for each individual function is written.

6.8.2 Debug Mechanism

Errors caused by faulty logic and coding mistakes are referred to as bugs. Finding and correcting these mistakes and errors that prevent the program from running and producing correct output is called debugging. Some common mistakes which cause program bugs are : mistakes in coding punctuation, incorrect operation codes, transposed characters, keying errors and failure to provide a sequence of instructions needed to process certain conditions.

The way of debugging the program code :

6.8.2.1 Runtime error

The program does something, but not as expected – a great way to make sure the code is getting executed.

6.8.2.2 Debugger

Debugging is the process of correcting or modifying the code in the program so that the program can build, run smoothly, act as expected and be easy to maintain later.

Example : // Activate debugging (change value from 0 to 1) for programmer convenience to check for error.

```
#define DEBUG_LISTS 1
```

```
#define DEBUG_LIST_LENGTHS_ONLY 1
```

6.9 Summary

This chapter outline the hardware and software requirements besides phases that involved in program development process. It also describe the important aspect for system coding, factors that contribute for program coding approach and modules in simulator. Last but not least, the type of coding style and debug mechanism which used in program coding.

7.1 Introduction

The main function of testing is to establish the presence of defects. Testing is partly an art, and it is well known that it is difficult to teach. Testing is not a purely technical activity, and it is not a purely scientific activity. The goal of system testing is to verify that the product is functioning according to the design, that it is performing all necessary functions and job correctly. The goal of system testing is to verify that the product is functioning according to the design, that it is performing all necessary functions and job correctly. The goal of system testing is to verify that the product is functioning according to the design, that it is performing all necessary functions and job correctly.

CHAPTER 7 :

SYSTEM TESTING

7.2 System Testing

System testing is the final stage of testing, where the entire system is tested.

7.2.1 **Objectives of System Testing** The main objectives of system testing are to verify that the system is functioning according to the design, that it is performing all necessary functions and job correctly. The goal of system testing is to verify that the product is functioning according to the design, that it is performing all necessary functions and job correctly.

7.2.2 **Steps in System Testing** The steps in system testing are to plan the test, execute the test, and report the results. The goal of system testing is to verify that the product is functioning according to the design, that it is performing all necessary functions and job correctly. The goal of system testing is to verify that the product is functioning according to the design, that it is performing all necessary functions and job correctly.

Chapter 7 : System Testing

7.1 Introduction

The main function of testing is to establish the presence of defect. Testing is performed to ensure that it is working correctly and efficiently and generally focused on two areas : internal efficiency and external effectiveness. The goal of *external effectiveness testing* is to verify that the simulator is functioning according to system design, that is performing all necessary functions and sub-functions. The goal of *internal testing* is to make sure that the computer code is efficient, standardized and well documented. Testing can be a labor-intensive process, due to its iterative nature. After simulator has been verified, it needs to be thoroughly tested to ensure that every component of the simulator is operating as it should and it is performing exactly in accordance with the requirements.

7.2 Testing Methodology

There are two main methodologies of testing : white-box and black-box testing.

7.2.1 White-box testing examines the internal structure of a program and attempts to test each logical case. White-box testing can be thought of as transparent box testing : the tester can see and test a specific section of code.

7.2.2 Black-box testing also known input/output-driven testing in which the tester views the program as a black box, and as such, the inner workings of the program are unknown. The main tool used in black-box testing is the specification of the program : attempts to determine what input causes

the output of the program to be different from what the specifications would require.

7.3 Type of Testing

7.3.1 Module Testing

It is also referred to unit testing and it focuses on verification of the smallest unit of system design - the module. Using the detailed design specification as a guide, important control paths are tested to uncover errors within the boundary of the module.

Module testing were done on :

- Input by user for start node (x,y) and end node (x,y)
 - to ensure that program could read the valid metric (1) for path and not (9) for wall as available path. At the same time, to ensure that result of output is shown in right way in MS Dos prompt.

Types of error occurred during module testing :

- **Algorithm error** - error in the assembly of program code results in the output display area
- **Syntax error** - innocent mistakes during keying in the program code.
- **Parameter passing error** - Data type of argument passed were different from the argument in method().

7.3.2 Integration Testing

Testing two or more modules or functions together with the intent of finding interface defects between the modules or functions. Testing completed at as a part of unit or functional testing, and sometimes, becomes its own standalone test phase. Integration testing can involve a putting together of groups of modules and functions with the goal of completing and verifying that the simulator meets the system requirements.

Integration testing were done on :

- Open and closed list, by activate the `DEBUG_LIST` and `DEBUG_LIST_LENGTHS_ONLY`. Activate debugging for programmer convenience to check for error. Zero (0) means not activate and one (1) means activate debugging.

```
#define DEBUG_LISTS 1
```

```
#define DEBUG_LIST_LENGTHS_ONLY 1
```

For each steps of searching, program will test and check appropriately with function : `GetOpenListStart()`, `GetOpenListNext()`, `GetClosedListStart()` and `GetClosedListNext()` to ensure the algorithm evaluation is correct and each node position, solution steps and search steps accurately obtained.

7.3.3 System Testing

It ensures that the simulator as a whole satisfies input and output specifications and that interfaces between modules, programs or subsystems are correct. Emphasis is placed on simulator access, security, performance, and recovery capabilities. The modules tested in the integration tested were tested

again as a complete program. The system testing will verify the accuracy of the simulator process, input and output to ensure it follows the design specification and the system's requirement.

7.4 Example Testing

- ❖ Test whether the simulator still can find the shortest path if user input start node metric (6,11) that has been block all around by wall (9).

```
#### STL A* SEARCH IMPLEMENTATION ####
```

```
Press 1 to enter the program or 0 to exit --> 1
```

```
Enter start node for column-x : 6
```

```
Enter start node for row-y : 11
```

```
Enter end node for column-x : 8
```

```
Enter end node for row-y : 12
```

```
Steps : 1
```

```
Open :
```

```
Open list has 0 nodes
```

```
Closed :
```

```
Closed list has 1 nodes
```

```
Steps : 2
```

```
Open :
```

```
Open list has 0 nodes
```

```
Closed :
```

```
Closed list has 0 nodes
```

```
Search terminated. Did not find goal state
```

```
SearchSteps : 2
```

```
Press 1 to enter the program or 0 to exit --> 0
```

```
Press any key to continue
```

Figure 7.1 : Example output that applied using Manhattan method

Result : Simulator cannot find available path because it does not program the node to cross over the corner of wall. The node only can move either by

horizontal or vertical only based on Manhattan method principles. Refer to *Figure 7.2*, whereby 1 is path/road and 9 is wall/obstacle.

```
// x→ column
// 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 // y→ row
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // 00
1, 9, 9, 1, 9, 9, 1, 1, 1, 9, 1, 9, 9, 9, 9, 9, 1, 1, 1, 1, // 01
1, 1, 9, 1, 1, 9, 9, 9, 1, 9, 1, 9, 1, 9, 1, 9, 9, 9, 1, 1, // 02
1, 9, 9, 1, 1, 9, 9, 9, 1, 9, 1, 9, 1, 9, 1, 9, 9, 9, 1, 1, // 03
1, 1, 1, 1, 1, 1, 9, 9, 1, 9, 1, 9, 1, 1, 1, 1, 9, 9, 1, 1, // 04
1, 1, 1, 1, 9, 1, 1, 1, 1, 9, 1, 1, 1, 1, 9, 1, 1, 1, 1, 1, // 05
1, 9, 9, 9, 9, 1, 1, 1, 1, 1, 1, 9, 9, 9, 9, 1, 1, 1, 1, 1, // 06
1, 1, 9, 9, 9, 9, 9, 9, 9, 1, 1, 1, 9, 9, 9, 9, 9, 9, 9, 1, // 07
1, 9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 1, // 08
1, 1, 1, 9, 9, 9, 9, 9, 9, 9, 1, 1, 9, 9, 9, 9, 9, 9, 9, 1, // 09
1, 1, 1, 1, 1, 1, 9, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // 10
1, 9, 9, 9, 9, 9, 9, 1, 9, 1, 9, 1, 9, 9, 9, 9, 9, 1, 1, 1, // 11
1, 9, 1, 9, 1, 9, 9, 9, 1, 9, 1, 9, 1, 9, 1, 9, 9, 9, 1, 1, // 12
1, 9, 1, 9, 1, 9, 9, 9, 1, 9, 1, 9, 1, 9, 1, 9, 9, 9, 1, 1, // 13
1, 9, 1, 1, 1, 1, 9, 9, 1, 9, 1, 9, 1, 1, 1, 1, 9, 9, 1, 1, // 14
1, 1, 1, 1, 9, 1, 1, 1, 1, 9, 1, 1, 1, 1, 9, 1, 1, 1, 1, 1, // 15
1, 9, 9, 9, 9, 1, 1, 1, 1, 1, 1, 9, 9, 9, 9, 1, 1, 1, 1, 1, // 16
1, 1, 9, 9, 9, 9, 9, 9, 9, 1, 1, 1, 9, 9, 9, 9, 9, 9, 9, 1, // 17
1, 9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 1, // 18
1, 1, 9, 9, 9, 9, 9, 9, 9, 9, 1, 1, 9, 9, 9, 9, 9, 9, 1, 1, // 19
```

*Figure 7.2 : Parking map [20*20]*

- ❖ By modify coding condition loop in function MapGetNode::GetSuccessor(), simulator now can search for shortest path from start node (6,11) to end node (8,12). The node find their path by moving diagonally, instead only moving horizontal and vertical.

- ❖ Change coding condition loop in function MapGetNode::GetSuccessor() and test whether the new modification will take different effect on program execution and result.

```
if( (GetMap( x+1, y-1 ) < 9)
    && !((parent_x == x+1) && (parent_y == y-1)))
{   astarsearch->AddSuccessor( NewNode );   }
```

```
if( (GetMap( x+1, y+1 ) < 9)
    && !((parent_x == x+1) && (parent_y == y+1)))
{   NewNode = MapSearchNode( x+1, y+1 );
    astarsearch->AddSuccessor( NewNode );   }
```

```
if( (GetMap( x-1, y-1 ) < 9)
    && !((parent_x == x-1) && (parent_y == y-1)))
{   NewNode = MapSearchNode( x-1, y-1 );
    astarsearch->AddSuccessor( NewNode );   }
```

```
if( (GetMap( x-1, y+1 ) < 9)
    && !((parent_x == x-1) && (parent_y == y+1)))
{   NewNode = MapSearchNode( x-1, y+1 );
    astarsearch->AddSuccessor( NewNode );   }
```



```

#### STL A* SEARCH IMPLEMENTATION ####

Press 1 to enter the program or 0 to exit --> 1

Enter start node for column-x : 6
Enter start node for row-y : 11

Enter end node for column-x : 8
Enter end node for row-y : 12
Steps : 1
Open :
Open list has 2 nodes
Closed :
Closed list has 1 nodes
Steps : 2
Open :
Open list has 3 nodes
Closed :
Closed list has 2 nodes
Steps : 3
Open :
Open list has 3 nodes
Closed :
Closed list has 3 nodes
Steps : 4
Open :
Open list has 0 nodes
Closed :
Closed list has 0 nodes
Search found goal state
Node position : (6, 11)
Node position : (7, 10)
Node position : (8, 11)
Node position : (8, 12)
Solution steps : 3
SearchSteps : 4

Press 1 to enter the program or 0 to exit --> 0
Press any key to continue

```

Figure 7.3 : Example output that applied not using Manhattan method

7.5 Summary

Chapter 7 : System Testing basically describe more about testing, type of testing and its methodology that used. There also shown an example on how testing is been done for this program simulator.

Chapter 8 : System Evaluation & Summary

8.1 Simulator Strength

After the simulator is developed, A* algorithm has achieved its main objective. The simulator is used to search for best shortest path from start node until to the end node by using A* algorithm. The search only done after the user define the start node metric (x,y) and end node metric (x,y). The program will read the coordinate input by user and refer to the map resource that already been prepared by programmer. This is to check whether the value is valid, means not a metric for wall (9) and the metric is within the weighted state space of map.

The simulator effectively able to find shortest path either by moving horizontal and vertical (using Manhattan method) or by moving diagonally (better shortest path) from start node to the end node. Coding program also provide good debugging function to check for errors as a convenient tools for programmer. The code program can be easily modify either to change the coding or adding new modules to the code.

8.2 Systems Limitation and Constraint

- The simulator is not program to take resource of map from the .txt file text input (Notepad). Programmer already provide the map that is declared as global data in findpath.cpp. This array of map can be access from any function in same or different file with the map helper functions GetMap(). This means hardcore coding because only programmer or system developer can modify or change the map and determine valid value for path (1) and wall (9). Only one map is provided and use in one time for the simulator reference; except if the programmer willing to go through hassle task building more constructive coding.

- Sometimes the simulator misread the metric (x,y) that point to wall (9) as the valid path (1) for node. The simulator still able to search for the shortest path although user define such wrong input.

8.3 Problem and solution

During the system requirement and analysis phase, a lot of study and research has been carried out. The problem faced during the analysis and requirement phase were not as crucial as during the implementation phase. A lot of modification and work cannot be carried out due to lack of knowledge in certain areas and time constraint. Below are some of the problems encountered :

8.3.1 Lack of Programming Experience

Problem :

It is a major drawback during the implementation of the project. The initial choice of using the Microsoft Foundation Class (MFC) Library as the GUI interface for the simulator had to be change to MS Dos prompt windows. This decision was made because of the complicated structure of coding, vast amount of classes and the complexity of using MFC in C++.

Solution :

As time was the main factor, MS Dos prompt windows was chosen as output environment due to its friendly and easy use style. Nevertheless, a lack of experience and skills in programming has been the major obstacle from completely achieving the whole project's objective.

8.3.2 Development Time Factor

Problem :

Small prior knowledge in A* algorithm and how pathfinding algorithm really works demand for lots of studies that need to be done and learn within a short span of time. Due to this factor also, certain features is not implemented in this project.

Solution :

However some of the obstacles were resolved by doing personal studies and research through the Internet.

8.4 Future Enhancement

Due to time limitation, not all of the target objectives and ideas could be incorporated in this project. Future enhancement is essential to make the system more up-to-date, interesting and dynamic. These factors are crucial to create an interest on the user to use the system.

Ideas for future enhancement :

- Create a user-friendly and colourful graphical user interface (GUI) by using Microsoft Foundation Class (MFC) Library that could interact more with the user.
- Enhanced the obstacles requirements; such as to Tough state (sandbank) or Very Tough state (vehicle blocking) in A* algorithm simulation whereby the code program must be manageable and flexible to be modify.
- The simulator that able to provide variation template of different mapping that based on real parking space terrain in real world.

- Implement this simulator in real-world environment that is in parking lot area through the collaboration with update technology such as GPRS mapping or roaming. Users that have handheld devices such as iPod, 2.5G or 3G handphone and notebook or computer laptop could easily connected with the automated parking system that implement A* algorithm for the pathfinding search in parking environment.

8.5 Summary

This chapter will cover and discuss the simulator strength and limitation, the simulator's problem and solution, and a few suggestions to enhance the simulator in the future.

Conclusion

After conducting analysis and testing, it is concluded that the project has achieved its main objective, searching for best shortest path in fastest time manner from start node (parking entrance) until the end node (empty parking space) . There are more research need to be done in developing the simulator. With the first step taken, enhancement can still be made in the future to this version of simulator. The simulator could be made more up to date, dynamic and detail.

As the project has to be done in a short period of time and a lot of technical issue arises and need to resolve, a few problems has been encountered. Solution has been sought during testing. Encountering with problem has been proven to be a valuable learning experience.

I learnt that a good knowledge of software development life cycle could accommodate a developer to manage their project smoothly. All five phases, requirement analysis, system design phase, system coding, testing and maintenance need to be followed accordingly in order to build a good system (simulator). To build a good simulator also require time, effort and patience.

One the most essential knowledge gained from this project is the technique on problem solving. I was also able to practice my skill in programming Visual C++ language and gain a sufficient knowledge on how to build a simple coding for pathfinding, how the A* algorithm works to search for the shortest path and a lot more.

This project has helped me a lot in recognizing my poor skill in time management, project management and communication. These experiences and knowledge gained would certainly help me to manage and organize any future project and will make me become a better programmer besides a better person.

REFERENCES

&

BIBLIOGRAPHY

References

Books and Research Documentation

Transportation Research Record 845 : Transportation System Management and Parking. Transportation Research Board, National Research Council. National Academy of Sciences. Washington, D.C., 1982.

Judea Pearl. (1984). *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Alan Bundy. (1997). *Artificial Intelligence Techniques*. 4th revised ed. Springer-Verlag.

Doran, J. and Michie, D. (1966). Experiments with the graph traverser program. Proc. Royal Society of London, 294(A) : 235-259.

Elaine Rich & Kevin Knight. (1991). *Artificial Intelligent*. 2nd Ed. McGraw-Hill.

Peter E. Hart, Nils J. Nilsson & Bertram Raphael. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on System Sciences and Cybernetics, SSC-4(2) : 100-107.

Thomas A. Standish. (1995). *Data Structures, Algorithms & Software Principles in C*. Addison Wesley.

Robert Sedgewick. (1988). *Algorithms*. Addison-Wesley.

Jacob Shapiro, Jerry Waxman & Dany Nir. (1992). Level Graphs and Approximate Shortest Path Algorithm. Networks, vol. 22, pp 691-717. John Wiley & Sons Inc.

James W. Lark, Chelsea C. White & Kirsten Syverson. (1995). A Best-First Search Algorithm Guided by a Set-Valued Heuristic. IEEE, Systems, Man and Cybernetics, vol. 25, no. 7, pp. 1097-1101.

Chai. Ian & White. Jonathan David. (2002). *Structuring Data & Building Algorithms*. McGraw-Hill.

Deitel. Harvey M. & Deitel. Paul J. (2003). *C++ How To Program*. 4th Ed. Prentice Hall.

Journals and Articles From The Internet

Jagdev Singh Sidhu & Pauline S.C. Ng. "MAA : Five percent vehicles sales growth within reach". Star Online.

"Amit's Thoughts on Path Finding and A-Star".

(URL - <http://theory.stanford.edu/~amitp/GameProgramming>), 28/08/2004.

Patrick Lester. "A* Path Findings for Beginners".

(URL - <http://www.policyalmanac.org/games/aStarTutorial.htm>), 04/07/2004.

"System Requirements To Use Microsoft Visual Studio 6.0".

(URL - <http://msdn.microsoft.com/vstudio/previous/vs6/features/specifications.aspx>), 28/08/2004.

"What's New in Visual C++ Version 6.0".

(URL - <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccedit98/HTML/verefwhatsnewforvisualcversion6.0.asp>), 28/08/2004.

Microsoft Foundation Class Library (MFC).

(URL - http://searchvb.techtarget.com/sDefinition/0,,sid8_gci214094,00.html), 28/08/2004.

"High-Level Languages".

(URL - <http://www.daniweb.com/techtalkforums/thread1729.html>)

(URL: <http://www.daniweb.com/techtalkforums/showthread.php?t=8908&goto=nextnewest>), 28/08/2004.

Lim, Audrey. "The Truths About Malaysians".

(URL : http://www.thingsasian.com/goto_article/article.1872.html), 30/08/2004.

A* Algorithm Pseudocode.

(URL : <http://www.geocities.com/jheyesjones/pseudocode.html>), 1/09/2004.

NARPAC, Inc. – Robotic Parking (National Association To Restore Pride In America's Capital)

(URL : <http://www.narpac.org/METROPRK.HTM#robotic>), 1/09/2004.

Pathfinding : A Comparison of Algorithms.

(URL : http://www.cpcug.org/user/scifair/Preygel/Preygel.html#_Toc445443578), 1/09/2004.

Patrick Lester. Heuristics and A* Pathfinding.

(URL : <http://www.policyalmanac.org/games/heuristics.htm>), 2/09/2004.

Graph algorithms.

(URL : <http://encyclopedia.thefreedictionary.com/List%20of%20algorithms>), 4/09/2004.

Search algorithms.

(URL : <http://encyclopedia.thefreedictionary.com/List%20of%20algorithms>), 4/09/2004.

Genetic Algorithms.

(URL : <http://encyclopedia.thefreedictionary.com/Genetic%20Algorithms>), 4/09/2004.

Hill climbing.

(URL : <http://encyclopedia.thefreedictionary.com/Hill%20climbing>), 4/09/2004.

Pathfinding using the A star method.

(URL : <http://www.heni-online.de/libkdegames/pathdoc/index.html>), 6/09/2004.

Thomas Grubb's Delphi Pathfinding demo.

(URL : <http://www.riversoftavg.com/downloads.htm>), 6/09/2004.

A* algorithm tutorial.

(URL : <http://www.geocities.com/jheyesjones/astar.html>), 2/09/2004.

James Matthews. A* for the Masses.

(URL : <http://www.generation5.org/content/2000/astar.asp>), 2/09/2004.

(URL : <http://encyclopedia.thefreedictionary.com/A-star%20algorithm>), 19/07/2004.

Short Description of A*.

(URL : <http://www-cs-students.stanford.edu/~amitp/Articles/AStar5.html>), 19/07/2004.

Beginners Guide to Pathfinding Algorithms.

(URL : <http://ai-depot.com/Tutorial/PathFinding.html>), 21/07/2004.

Problem Description.

(URL : <http://ai-depot.com/BotNavigation/Path.html>), 21/07/2004.

Patrick Lester. Heuristics and A* Pathfinding.

(URL : <http://www.policyalmanac.org/games/heuristics.htm>), 25/07/2004.

Patrick Lester. Using Binary Heaps in A* Pathfinding.

(URL : <http://www.policyalmanac.org/games/binaryHeaps.htm>), 30/07/2004.

James Matthews. A* Explorer Version 2.0 (16th July, 2002).

(URL : <http://www.generation5.org/>), 3/1/2005.

Bibliography

Tee Wee Jing. (2002). *A Genetic Algorithm Solution To The Shortest Path Problem In OSPF & MPLS*. Master Thesis. University of Malaya.

Wan Hoong Thai. (May, 2002). *Using A* Algorithm For Solving Optimal Path Problem In Road Network*. Master Thesis. University of Malaya.

Vikneswaran Veerasamy. (2003). *Intelligent Map Guider*. BSc Thesis. University of Malaya.

A- Introduction Manual

In order to understand the normal operation of the equipment and

- Understand the structure

- Understand the function

- Understand the operation

APPENDIX A :

USER MANUAL

User Manual

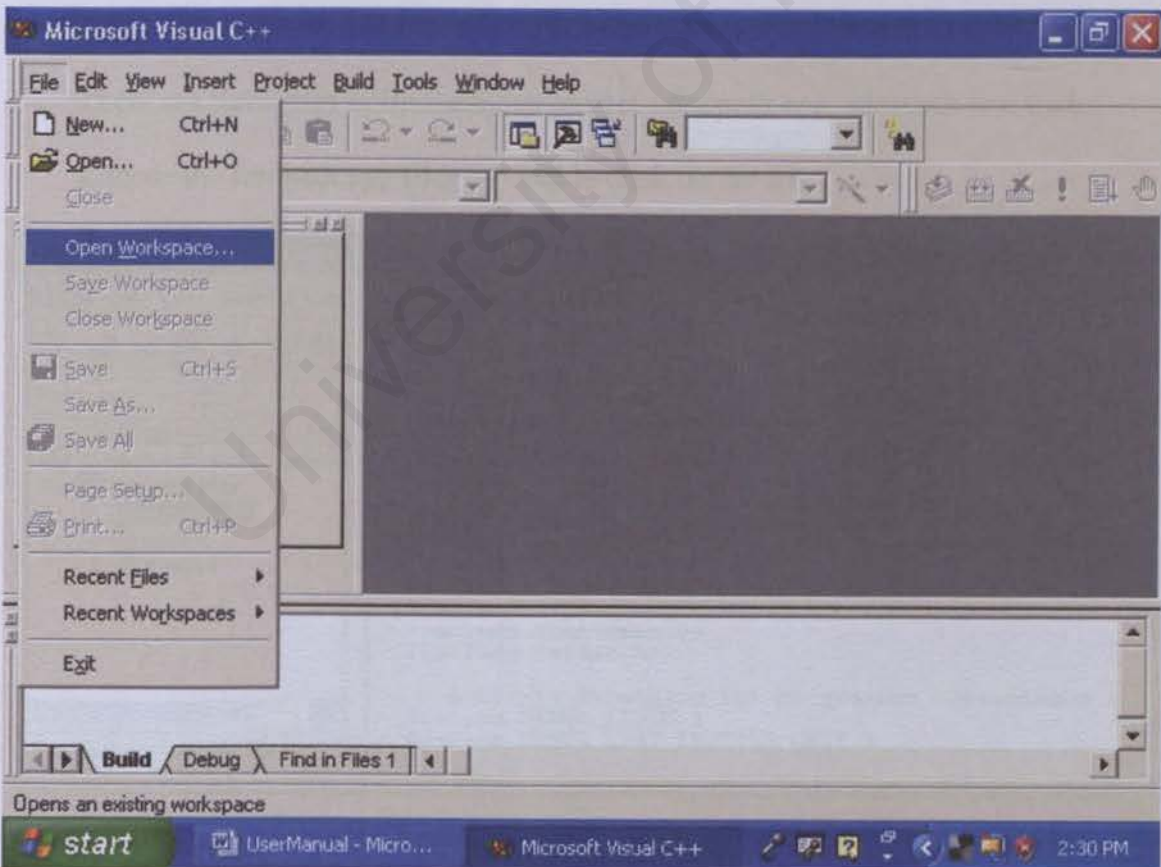
A - Installation Manual

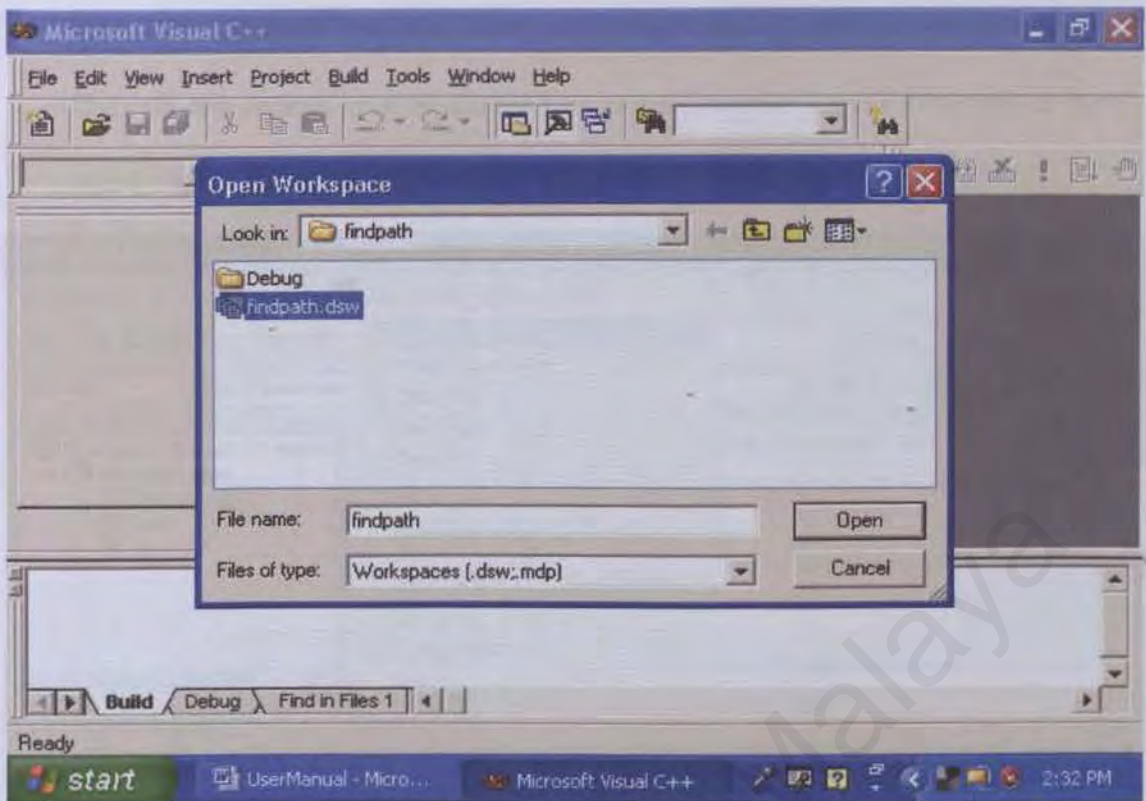
In order to use the simulator, the minimum requirements of your computer are :

- Windows 2000 and above
- Microsoft Visual C++ 6.0
- MSdos prompt windows

B - How To Use The Simulator of Shortest Path Using A-Star Algorithm

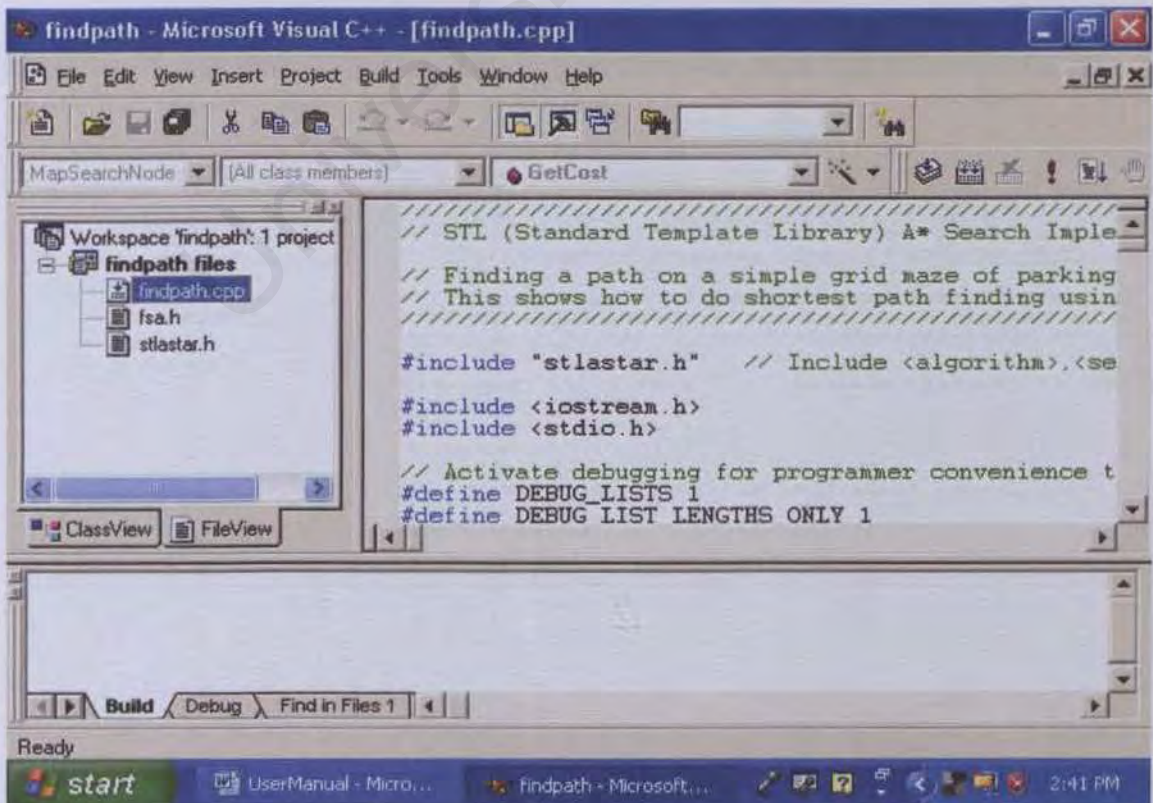
- 1 - Open the Microsoft Visual C++ 6.0 application and then open the findpath.dsw workspace file from a findpath folder.



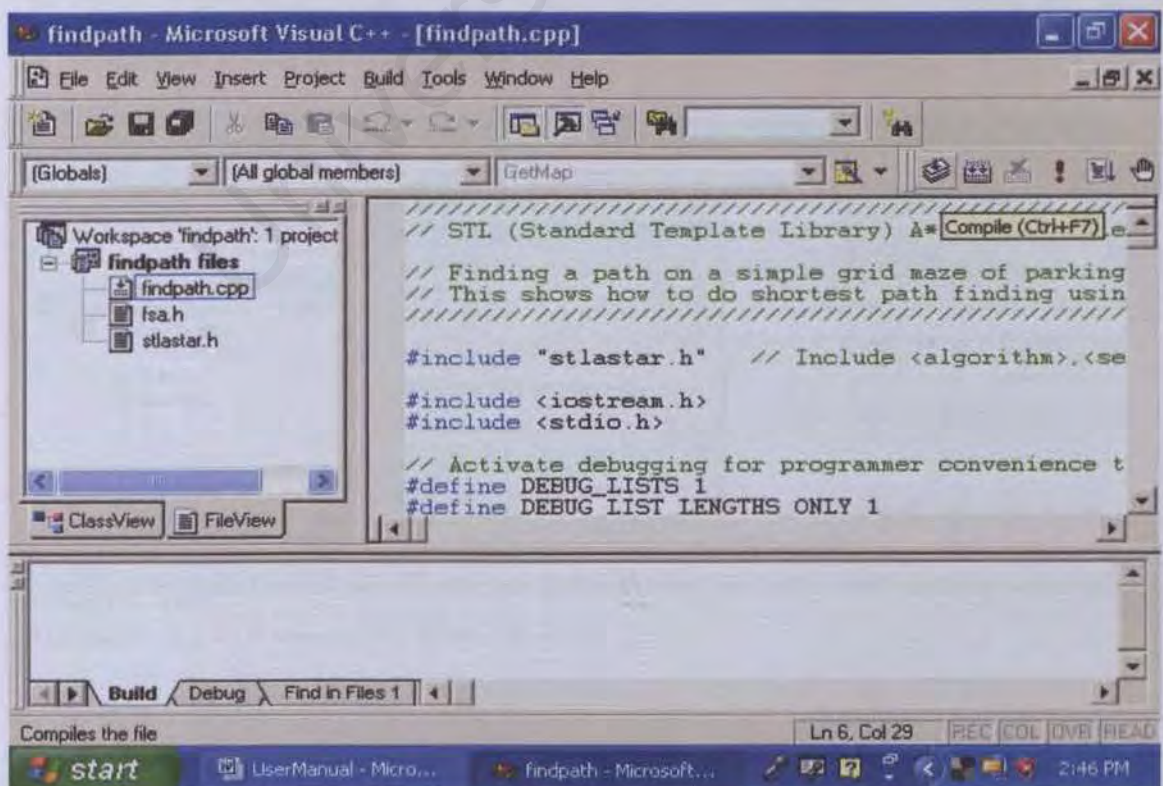
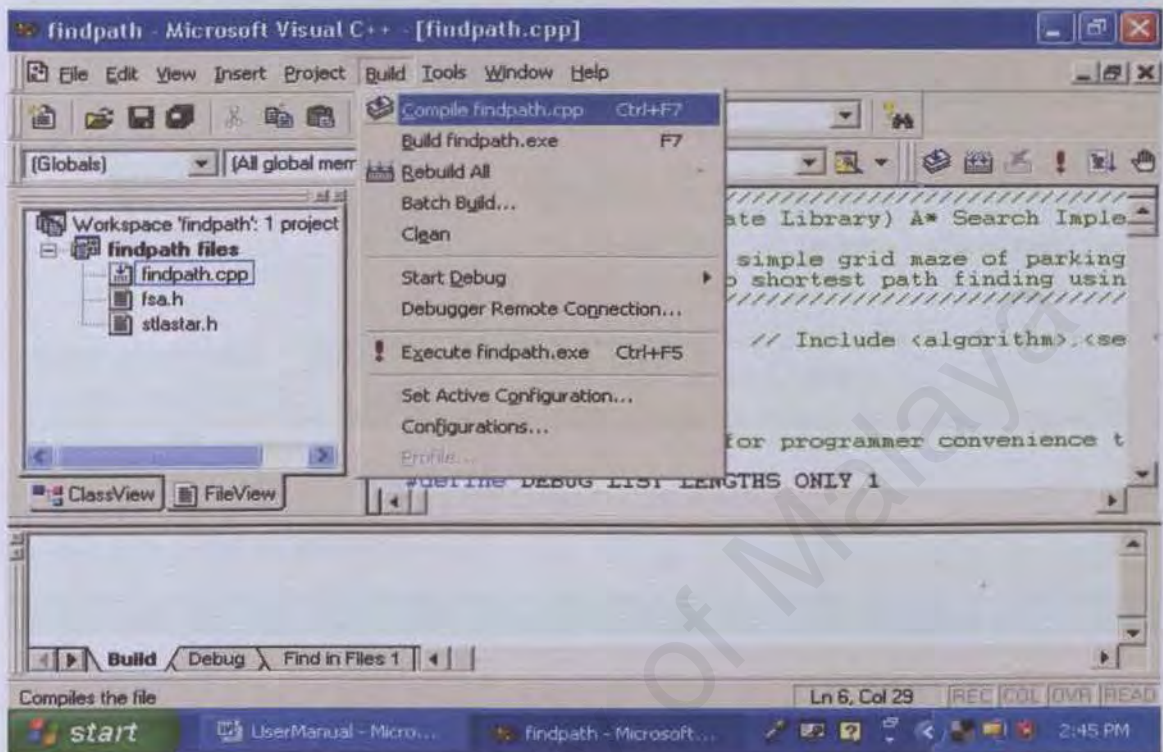


2 - There are three files in findpath.dsp project : findpath.cpp, stlastar.h and fsa.h.

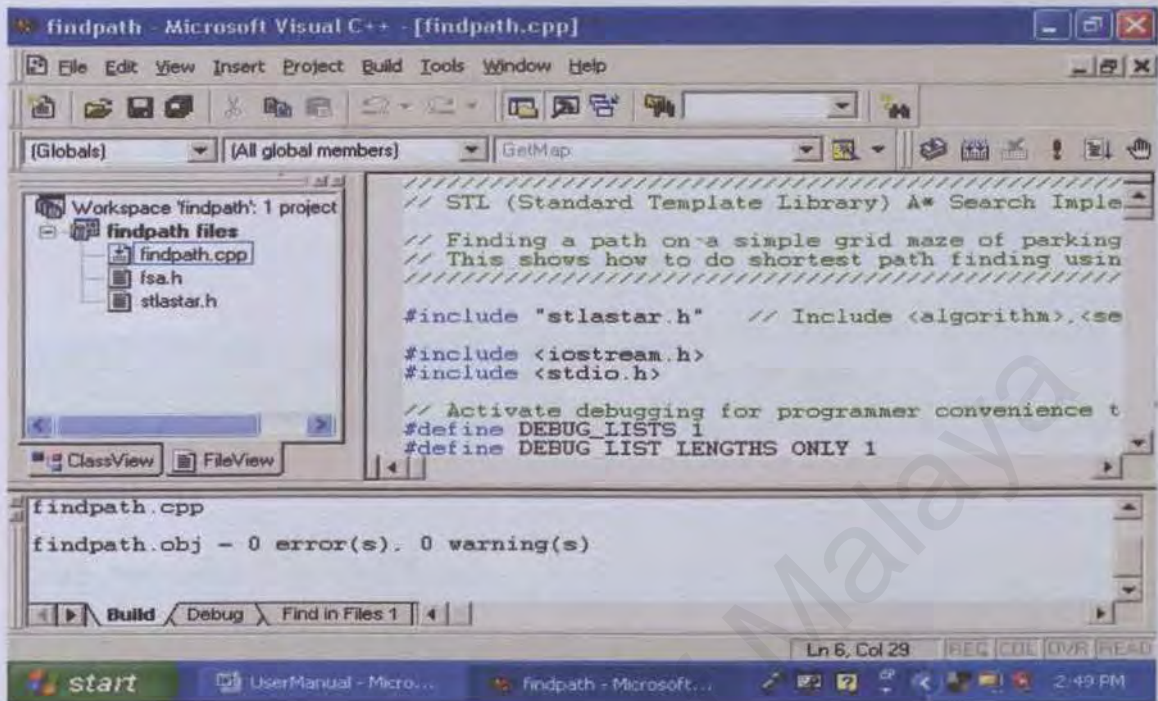
Choose the findpath.cpp files by double-click on the file.



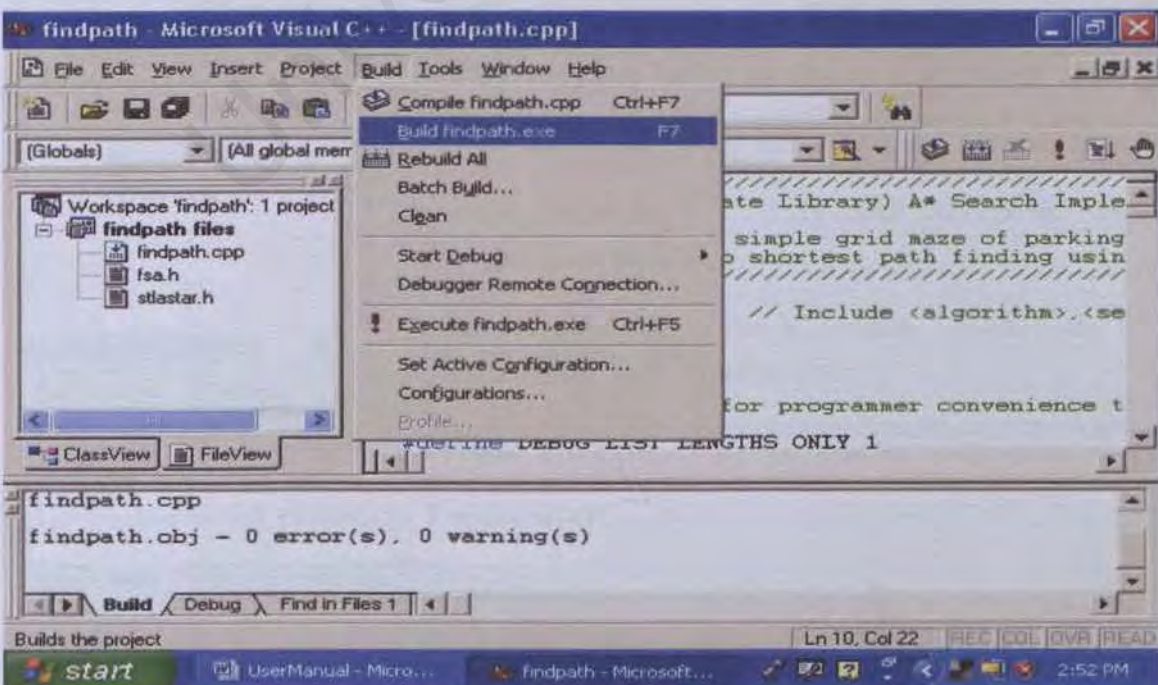
- 3 - To compile findpath.cpp, you either can go to Build menu and choose Compile findpath.cpp option, or press Ctrl+F7, or press Compile (Ctrl+F7) button on the upper-right side Microsoft Visual C++ 6.0 application.



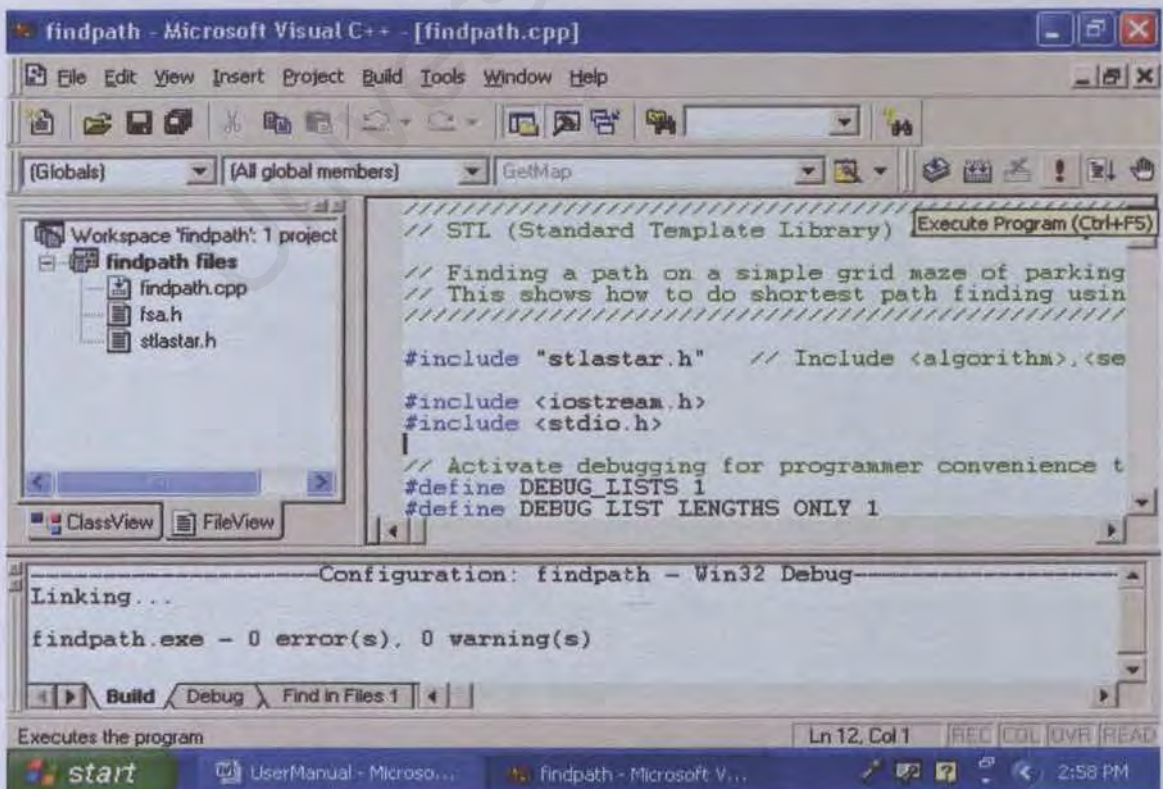
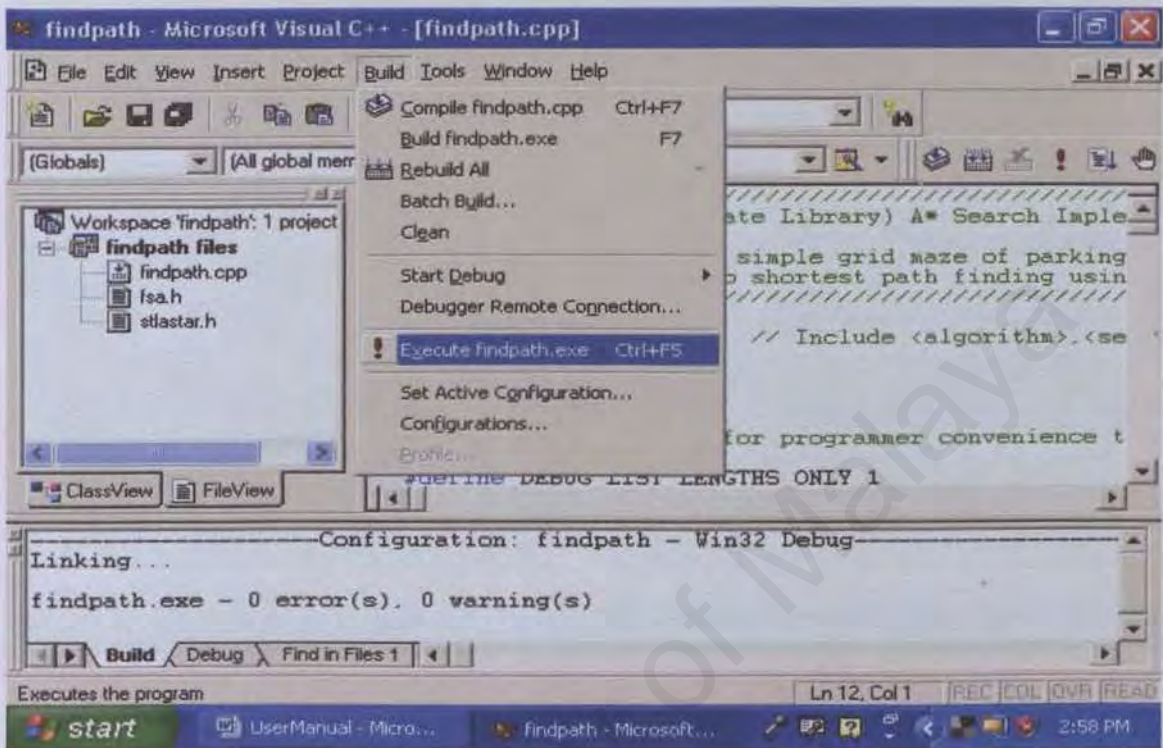
- 4 - After compile the program, you will get this result if there's no error in code program.



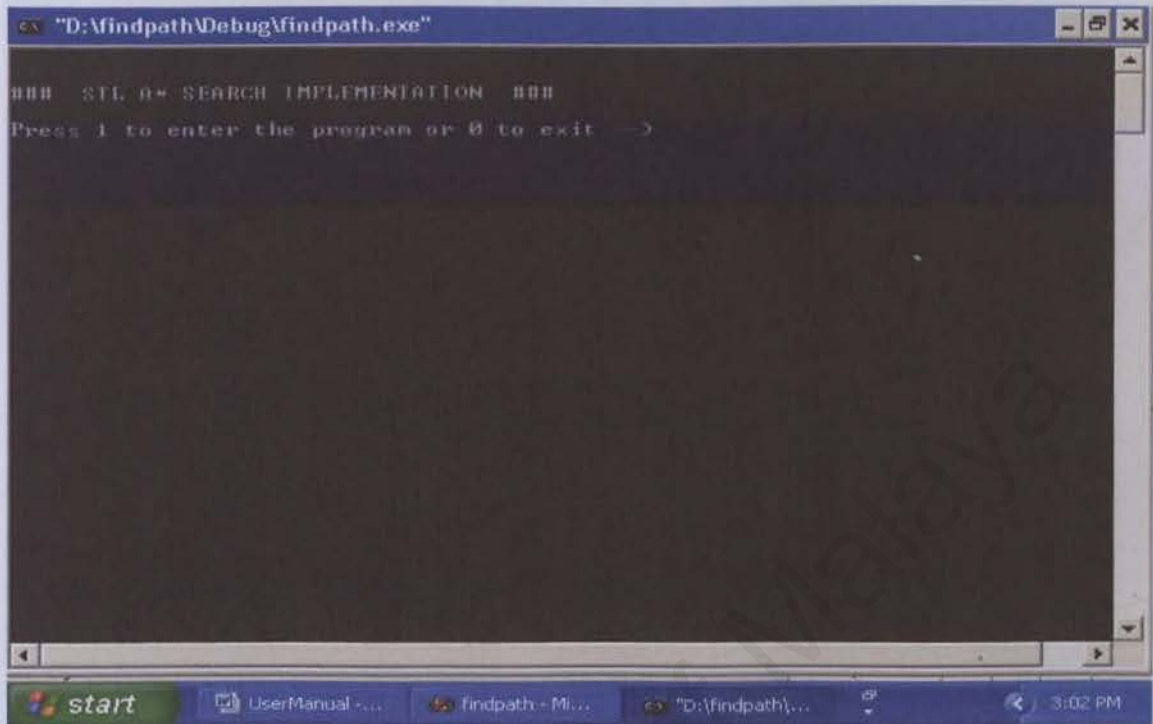
- 5 - To build findpath.cpp, you either can go to Build menu and choose Build findpath.exe option, or press F7, or press Build (F7) button on the upper-right side Microsoft Visual C++ 6.0 application.



- 7 - To run/execute findpath.cpp, you either can go to Build menu and choose Execute findpath.exe option, or press Ctrl+F5, or press Execute Program (Ctrl+F5) button on the upper-right side Microsoft Visual C++ 6.0 application.

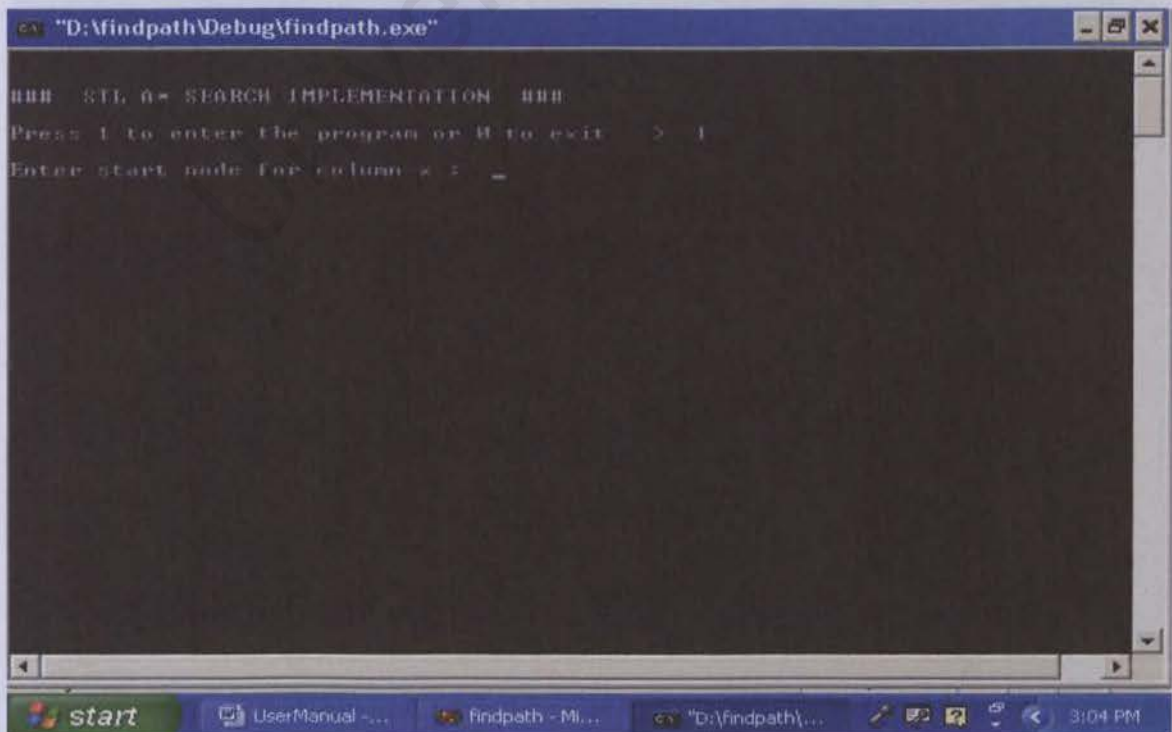


- 8 - After run/execute the program, you will get this result if there's no error in code program when you compile and build the program before.



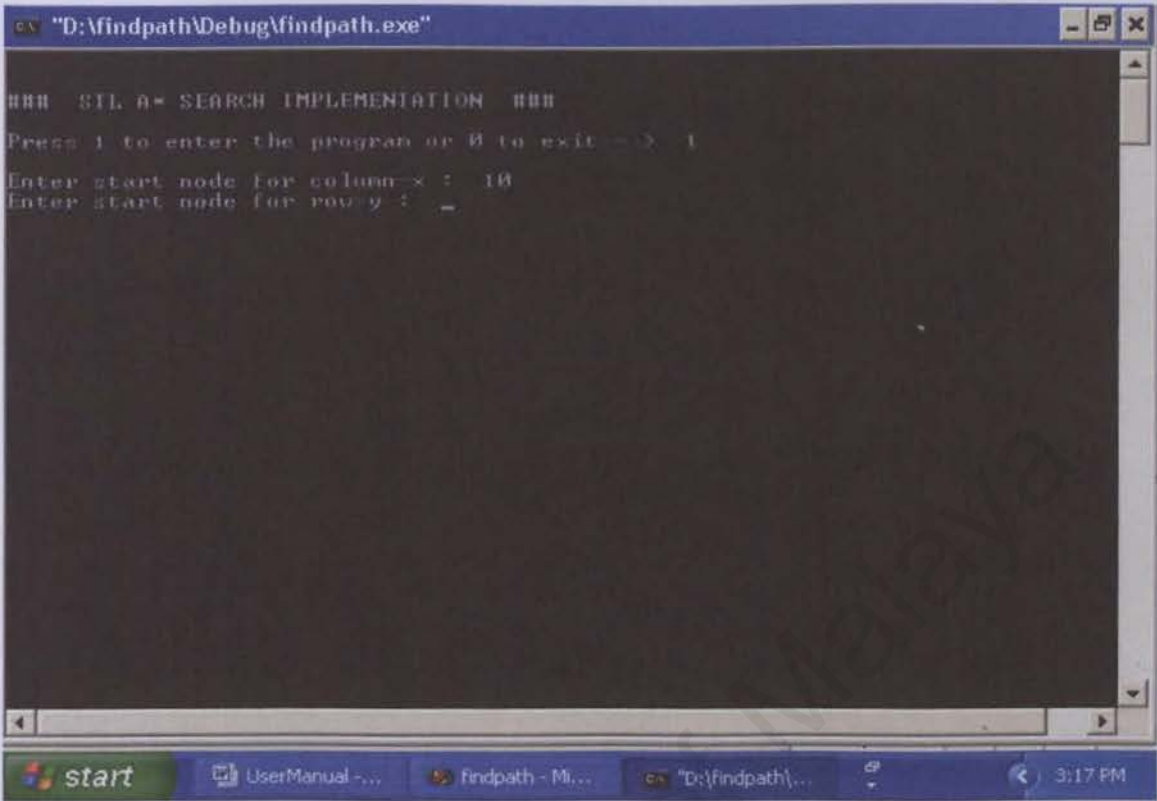
```
### STL: a* SEARCH IMPLEMENTATION ###
Press 1 to enter the program or 0 to exit ->
```

- 9 - Press 1 if you want to enter the program or press 0 if you want to exit from the program.

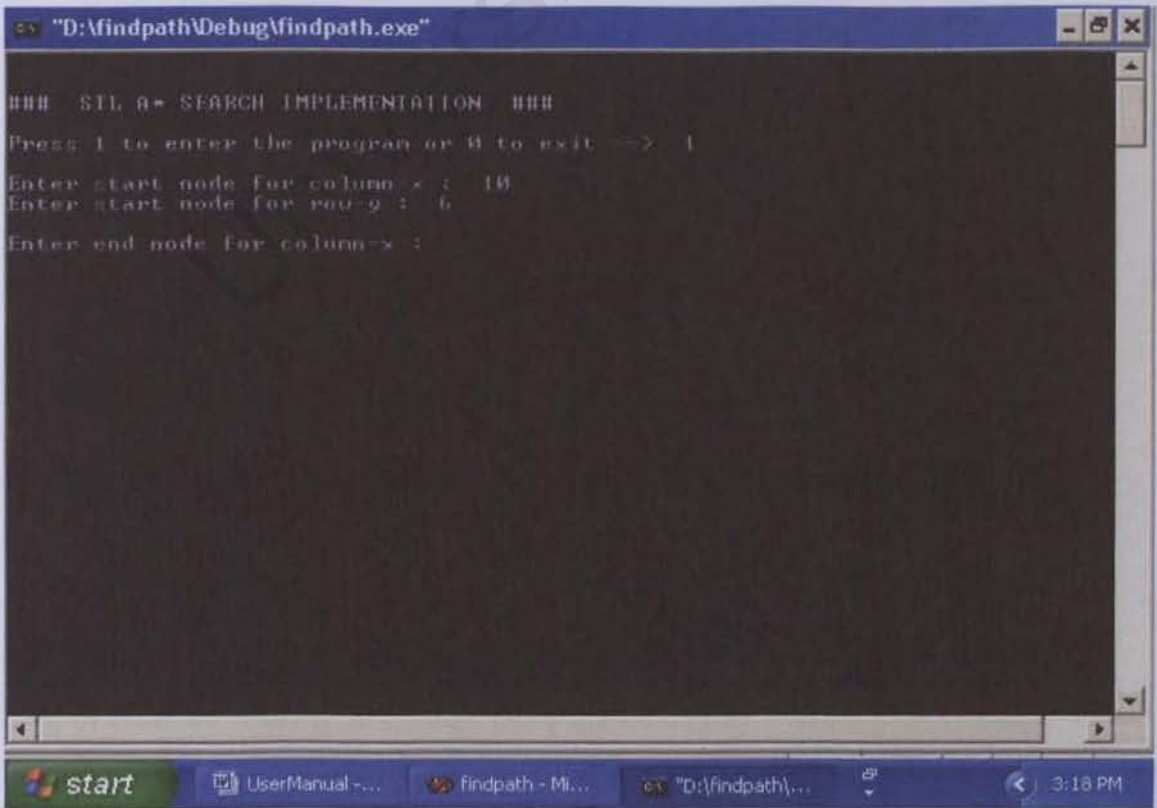


```
### STL: a* SEARCH IMPLEMENTATION ###
Press 1 to enter the program or 0 to exit -> 1
Enter start node for column x :
```

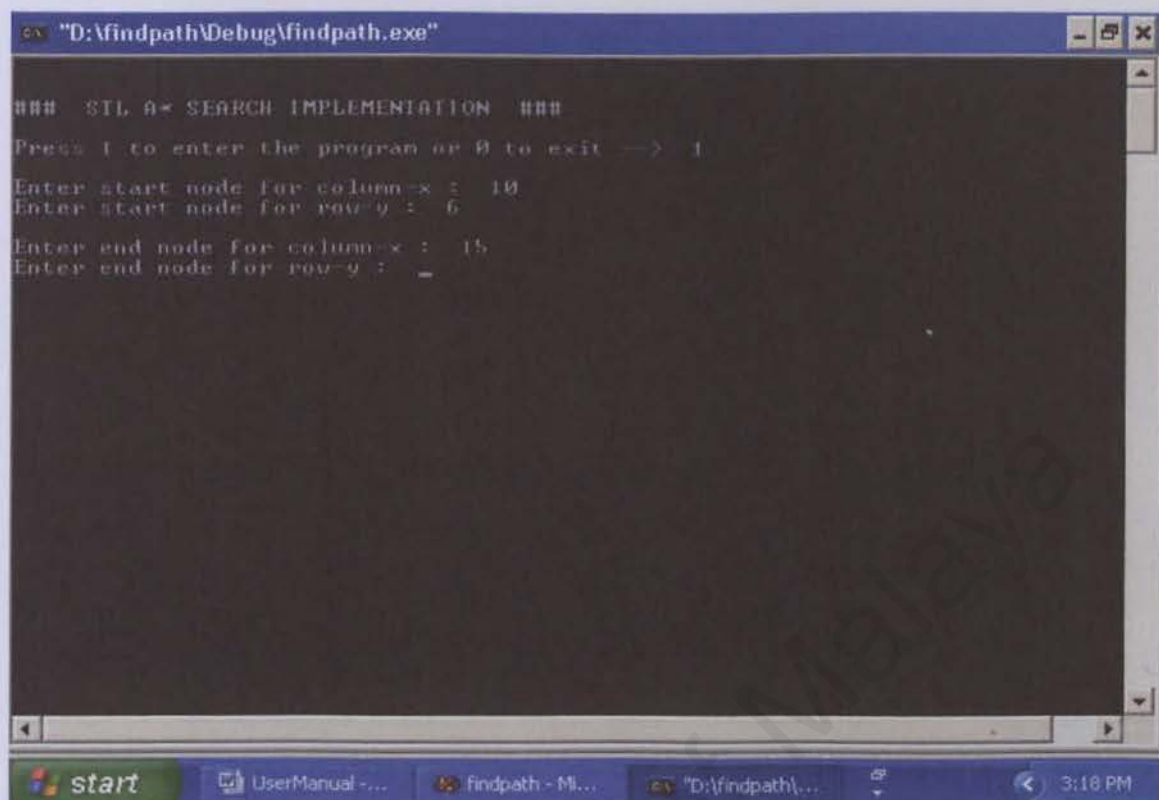

10 - Enter value of column-x for start node.



11 - Next enter value of row-y for start node.

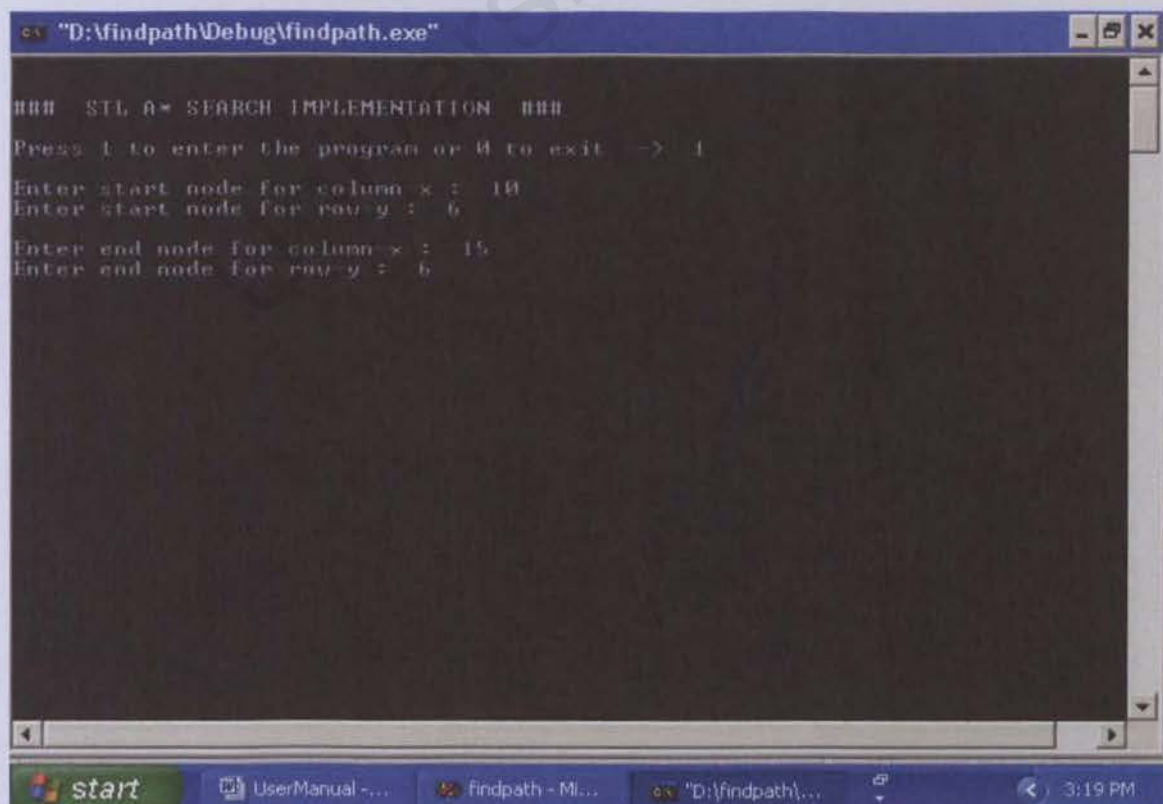


12 - Then enter value of column-x for end node.



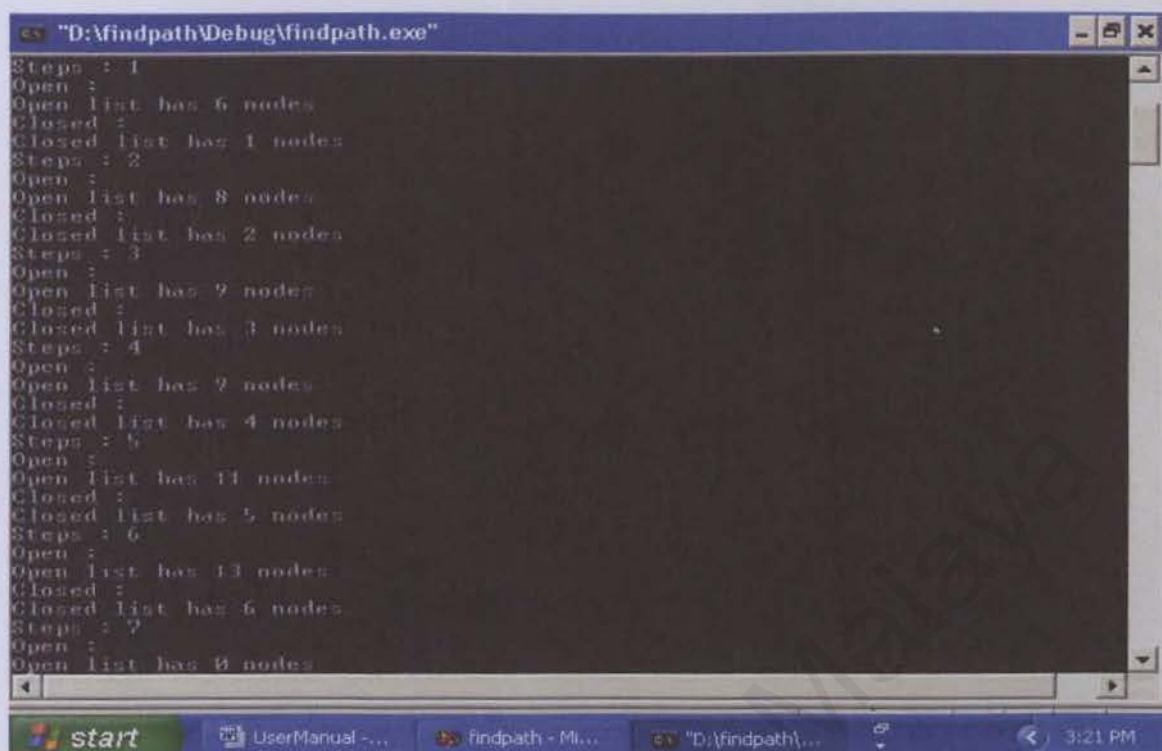
```
### STL A* SEARCH IMPLEMENTATION ###
Press 1 to enter the program or 0 to exit -> 1
Enter start node for column-x : 10
Enter start node for row-y : 6
Enter end node for column-x : 15
Enter end node for row-y :
```

13 - Now, enter value of row-y for end node.



```
### STL A* SEARCH IMPLEMENTATION ###
Press 1 to enter the program or 0 to exit -> 1
Enter start node for column-x : 10
Enter start node for row-y : 6
Enter end node for column-x : 15
Enter end node for row-y : 6
```

14 - When you press enter, you will get this result.

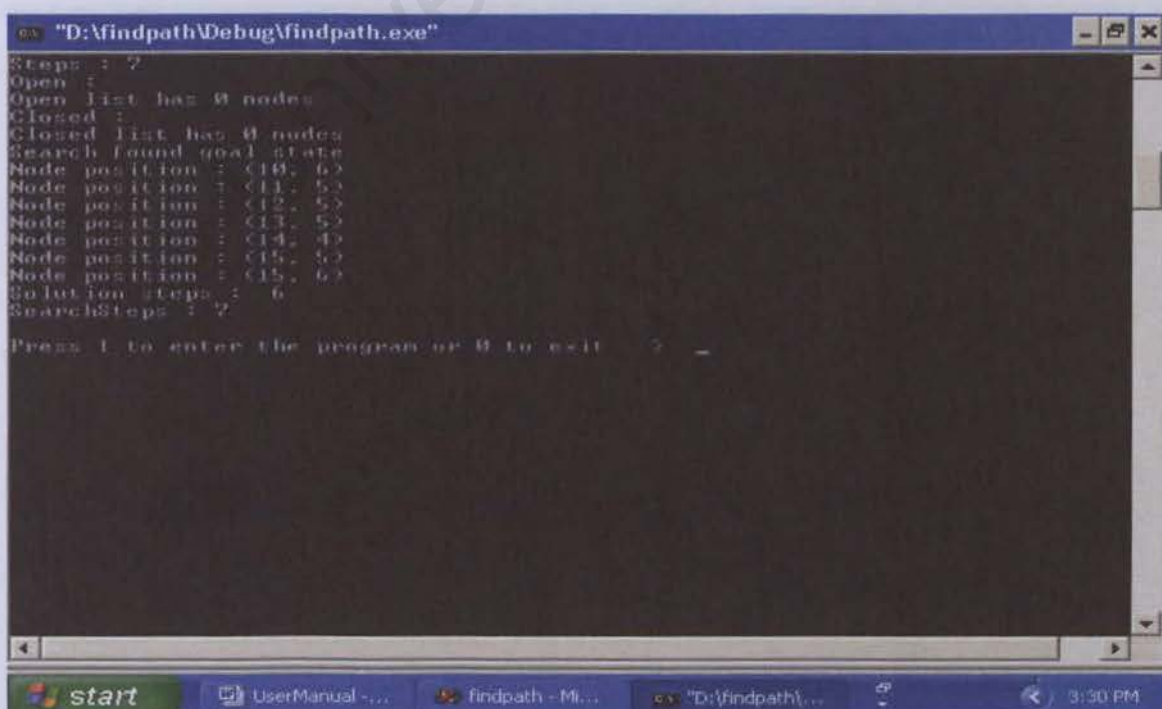


```

D:\findpath\Debug\findpath.exe
Steps : 1
Open :
Open list has 6 nodes
Closed :
Closed list has 1 nodes
Steps : 2
Open :
Open list has 8 nodes
Closed :
Closed list has 2 nodes
Steps : 3
Open :
Open list has 2 nodes
Closed :
Closed list has 3 nodes
Steps : 4
Open :
Open list has 2 nodes
Closed :
Closed list has 4 nodes
Steps : 5
Open :
Open list has 11 nodes
Closed :
Closed list has 5 nodes
Steps : 6
Open :
Open list has 13 nodes
Closed :
Closed list has 6 nodes
Steps : 7
Open :
Open list has 8 nodes

```

15 - The code program will evaluate and search for shortest path and show each step of searching using A* algorithm, the open and closed list activities, each node position in shortest path, number of solution steps and search steps taken.

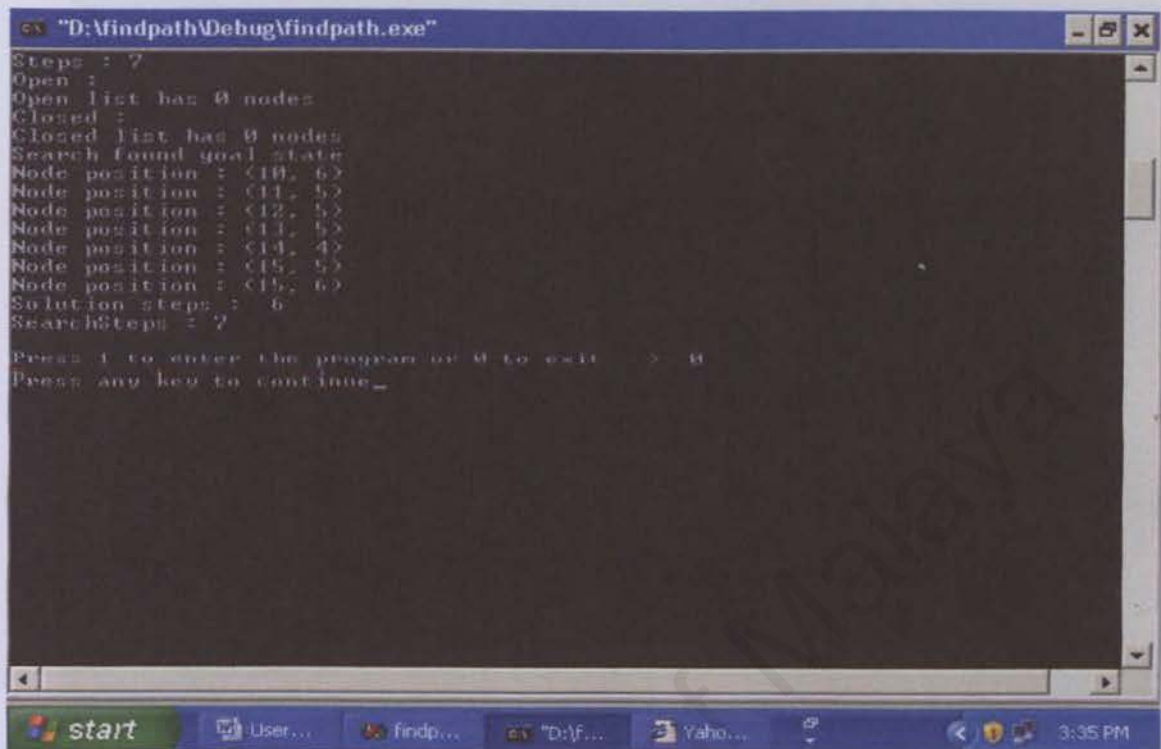


```

D:\findpath\Debug\findpath.exe
Steps : 7
Open :
Open list has 8 nodes
Closed :
Closed list has 8 nodes
Search found goal state
Node position : <10, 6>
Node position : <11, 5>
Node position : <12, 5>
Node position : <13, 5>
Node position : <14, 4>
Node position : <15, 5>
Node position : <15, 6>
Solution steps : 6
SearchSteps : 7
Steps : 8
Open :
Open list has 0 nodes
Closed :
Closed list has 0 nodes
Press 1 to enter the program or 0 to exit

```

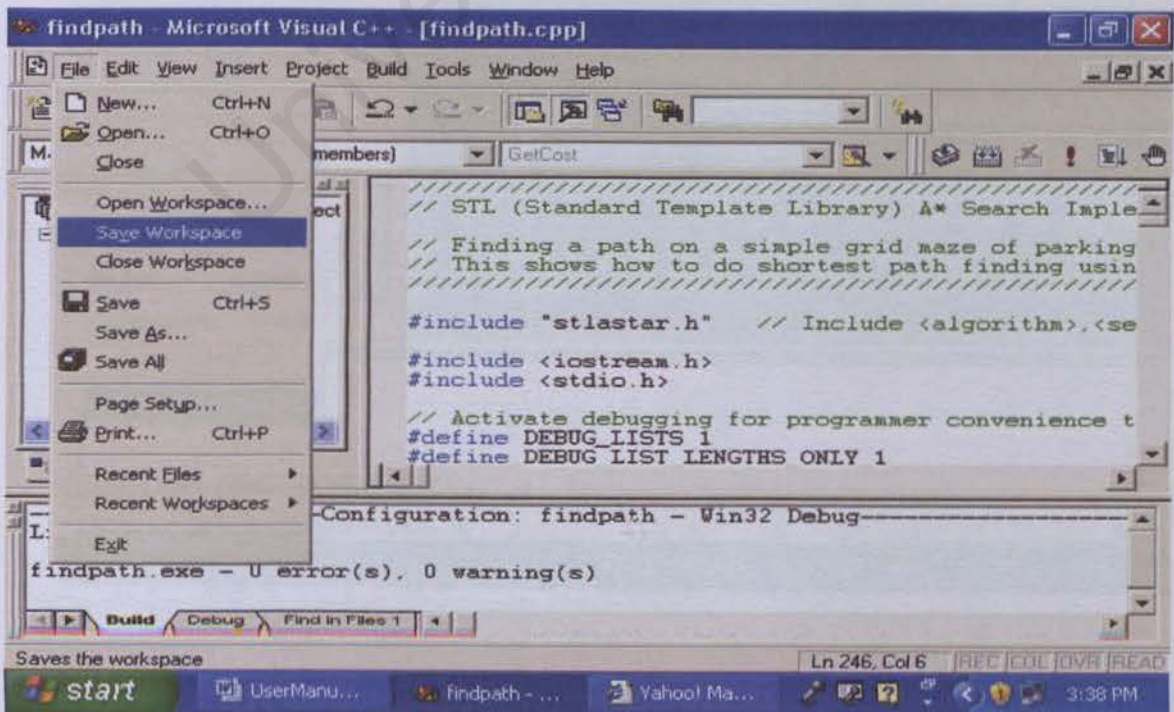

- 16 - To exit from the program, press 0 and enter. You will go back to the findpath.dsw workspace in Microsoft Visual C++ 6.0 environment.



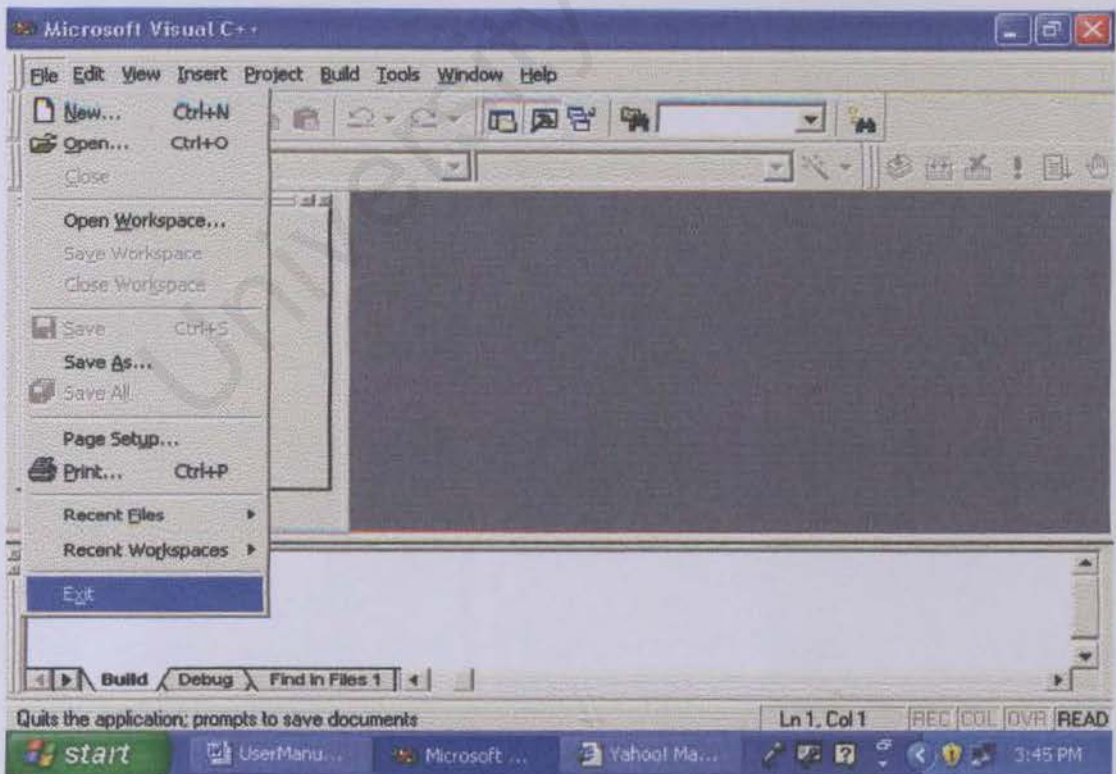
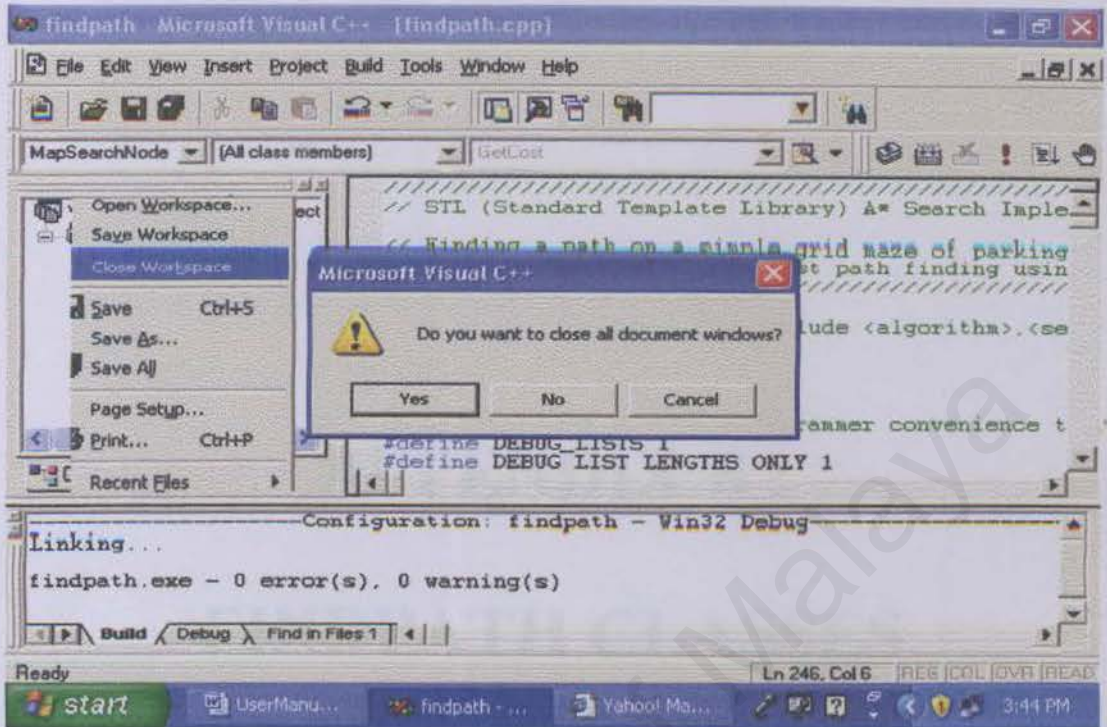
```
"D:\findpath\Debug\findpath.exe"
Steps : 7
Open :
Open list has 0 nodes
Closed :
Closed list has 0 nodes
Search found goal state
Node position : (10, 6)
Node position : (11, 5)
Node position : (12, 5)
Node position : (13, 5)
Node position : (14, 4)
Node position : (15, 5)
Node position : (15, 6)
Solution steps : 6
SearchSteps : 7

Press 1 to enter the program or 0 to exit > 0
Press any key to continue...
```

- 17 - Save the findpath.dsw workspace file by going to File menu and choose Save Workspace.



18 - Exit (close) from the Microsoft Visual C++ 6.0 application, go to File menu and choose Close Workspace option and Exit option.



APPENDIX B :

FINDPATH CLASSES

// findpath.dsw classes

- 1- AStarSearch<class UserState> --> in stlastar.h header file
- 2- FixedSizeAllocator<class USER_TYPE> --> in fsa.h header file
- 3- MapSearchNode --> in findpath.cpp file

4- GLOBALS (in findpath.cpp file) :

- GetMap(int x, int y)
- main (int argc, char *argv[])
- map

//

AStarSearch<class UserState>

1- HeapCompare_f

- operator() (const AStarSearch<UserState>:Node *x, const
AStarSearch<UserState>:Node *y)

2- Node

Member functions : Node()

Data members : child

(6) f
g
h
m_UserState
parent

- Member functions (23) :

- AddSuccessor(UserState &State)
- AllocateNode()
- AStarSearch(int MaxNodes=1000)
- CancelSearch()
- FreeAllNodes()
- FreeNode(AStarSearch<UserState>.Node *node)
- FreeSolutionNodes()
- FreeUnusedNodes()
- GetClosedListNext(float &f, float &g, float &h)
- GetClosedListNext()
- GetClosedListStart(float &f, float &g, float &h)
- GetClosedListStart()
- GetOpenListNext()
- GetOpenListNext(float &f, float &g, float &h)
- GetOpenListStart()
- GetOpenListStart(float &f, float &g, float &h)
- GetSolutionEnd()
- GetSolutionNext()
- GetSolutionPrev()
- GetSolutionStart()
- GetStepCount()
- SearchStep()
- SetStartAndGoalStates(UserState &Start, UserState &Goal)

- Data members : m_MaxElements
- (4) m_pFirstFree
- m_pFirstUsed
- m_pMemory

1- FSA_ELEMENT

- Data members : pNext
- (3) pPrev
- UserType

MapSearchNode

- Member functions (8) :
 - GetCost(MapSearchNode &successor)
 - GetSuccessors(AstarSearch<MapSearchNode> *astarsearch, MapSearchNode *parent_node)
 - GoalDistanceEstimate(MapSearchNode &nodeGoal)
 - IsGoal(MapSearchNode &nodeGoal)
 - IsSameStateMap(MapSearchNode &rhs)
 - MapSearchNode()
 - MapSearchNode(unsigned int px, unsigned int py)
 - PrintNodeInfo()
- Data members : x
- (2) y

APPENDIX C :
PROJECT SCHEDULE
– GANTT CHART

7/10

7/28

8/7

9/4

10/1

10/2

7/1

244 days

PROJECT START : Thu 7/1/04
EXPECTED PROJECT END :
Tue 3/1/05

11/11/2016

Y O U R S E L F

